Markt&Technik

Manfred Kohlen

GRAFIK AUF DEM AMIGA

Grundwissen zur Computergrafik
* Grafik-Programmierung mit Amiga-BASIC * Die fantastischen Grafik-Fähigkeiten der Hardware * Der Aufbau des Amiga-Betriebssystems * Animations-, CAD- und Malprogramme im Überblick

Manfred Köhlen

Grafik auf dem Amiga

Grundwissen zur Computergrafik
- Grafik-Programmierung mit Amiga-BASIC
- Die fantastischen Grafik-Fähigkeiten der Hardware
- Der Aufbau des Amiga-Betriebssystems
- Animations-, CAD- und Malprogramme im Überblick

Markt&Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Kohlen, Manfred:

Grafik auf dem Amiga : Grundwissen zur Computergrafik ; Grafikprogrammierung mit Amiga-BASIC, d. fantast. Grafik-Fähigkeiten d. Hardware, d. Aufbau d. Amiga-Betriebssystems, Animations-, CAD- u. Malprogramme im Überblick / Manfred Kohlen. – Haar bei München : Markt-und-Technik-Verlag, 1987.
ISBN 3-89090-236-7

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
90 89 88 87

ISBN 3-89090-236-7

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany

Inhaltsverzeichnis

Buchteil 2

Buchteil 5



Vorwort

Vor langer Zeit (»Es war einmal...«) gingen die ersten Gerüchte über einen neuen Computer um, der phantastische Grafikmöglichkeiten bieten sollte. Geplant und entwickelt wurde das Gerät während der Zeit des Booms der Telespiele in den USA unter dem Decknamen »Lorraine«. Um diesem Spieleboom gerecht zu werden und Möglichkeiten zu bieten, die kein anderes für Spiele gedachte Gerät bot, wurden Grafikfähigkeiten im Amiga vorgesehen, die denen der Spielhallengeräte in nichts nachstehen sollten. Die phantastischen Grafikfähigkeiten des Amiga verdanken wir in erster Linie also den Computerspielen.

Doch der Markt hatte sich geändert als der Amiga fertiggestellt war. Er wurde daraufhin weiterentwickelt und findet nun sowohl als Businesscomputer wie auch im Heimbereich Verwendung. Der Amiga ist ein professionelles System, das sich nicht hinter anderen PCs zu verstecken braucht. Seine Grafikfähigkeiten lassen sich zum Beispiel hervorragend für Businessgrafiken anwenden (Kuchen- oder Balkendarstellung, Gewinnfunktionen und ähnliches).

Aber auch für Nischenmärkte, wie zum Beispiel Design und Grafik, für die bisher wesentlich größere und teurere Geräte gekauft werden mußten, ist der Amiga hervorragend geeignet. Durch die außergewöhnlichen Grafiken, die man mit dem Amiga in erstaunlicher Geschwindigkeit zustandebringt, ist er zudem eine unwillkommene Konkurrenz für Anbieter professioneller CAD-Systeme und durch die bereits erhältliche zusätzliche Hardware sogar für die Arbeit in Videostudios geeignet.

Die Konzeption dieses Buches

Ich stand zunächst vor dem Problem, für welche Käuferschicht ich dieses Buch eigentlich schreiben sollte. Bei einem Allround-Computer wie dem Amiga weiß man nicht so recht, wie sich der Markt entwickeln wird. Daher entschloß ich mich nach langem Hin- und Herüberlegen zunächst, ein Buch zu schreiben, das möglichst alle Grafikanwendungen beinhaltet. Da der Amiga so viele Grafikfähigkeiten bietet wie kein anderer Computer dieser Preisklasse, würde dabei allerdings ein Mammutwerk von mindestens 1000 Seiten herausspringen. Und dafür hatte ich weder die nötige Zeit noch die Geduld. Was tun? Mir blieb die Alternative, entweder für erfahrene Programmierer zu schreiben – was schon Arbeit mit anderen Computern und gewisse Grundkenntnisse voraussetzt – oder für Anfänger. Daneben gibt es noch die Anwender, die lieber nur Informationen zur Arbeit mit professioneller Grafiksoftware haben wollen. Und dann gäbe es da noch die Hardwarespezialisten, und, und ... Schließlich kam ich auf den Gedanken, mich erst einmal dem Anfänger zu widmen, denn er stellt den größten Teil der Amiga-Gemeinde dar. Eventuelle weitere Bände kann man ja noch immer machen, und an einem weiteren Projekt arbeite ich bereits schon.

In der vorliegenden Form ist dieses Buch nun in mehrere Teile gegliedert, die unabhängig voneinander gelesen werden können. Wo Wissen aus anderen Buchteilen notwendig ist, wird im entsprechenden Kapitel immer ein Querverweis auf das benötigte Kapitel des anderen Buchteils gegeben. So können Sie immer nur das lesen und lernen, was Sie gerade benötigen, ohne unnötig in für Sie uninteressanten Kapiteln blättern zu müssen. Teil 1 dieses Buches führt in die grundsätzlichen Möglichkeiten und Grundlagen der Grafikprogrammierung einschl. Grundkenntnisse zum Thema Grafik sowie mathematische Grundlagen, die man für eine sinnvolle Anwendung in eigenen Programmen und für das Umschreiben der Programme auf andere Computer einfach braucht, werden hier vermittelt. Im Teil 2 werden anschließend die speziellen Fähigkeiten des Amiga in grafischer Hinsicht nicht nur vorgestellt, sondern auch deren Funktionsweise und die zugrundeliegende Hardware erklärt. Die Programmierung der Grafik will ich im Buchteil 3 anhand einer Programmiersprache erklären, die jedem Amiga-Besitzer zur Verfügung

steht – AmigaBasic von Microsoft. Dabei werden wir – Sie als Schüler, ich als Lehrer – gemeinsam am Punkt Null anfangen. Wir gehen von den einfachen Grafikbefehlen langsam zu immer schwierigeren Themen vor. Dabei werden jedoch nicht nur die einfachen Befehle und ihre Anwendungsmöglichkeiten gezeigt, sondern auch Techniken zur schnellen Animation von Objekten, ja sogar von ganzen Bildern, vorgestellt. Da für bestimmte Anwendungen Basic zu langsam wird, werden Sie sich sicherlich eines Tages mit der Programmierung in anderen Sprachen beschäftigen wollen. Dazu müssen Sie jedoch erst einmal den Aufbau des Betriebssystems des Amiga kennen.

Buchteil 4 vermittelt die entsprechenden Grundlagen. Sie erfahren, wie das Betriebssystem des Amiga aufgebaut ist, was für Grafikmöglichkeiten es bietet, und wie sie funktionieren. Zur intensiveren Systemprogrammierung des Amiga-Betriebssystems kann Sie dieses Buch aber nicht führen. Dazu empfehle ich Ihnen, vorher eine andere Programmiersprache zu lernen. (Empfehlungen hierzu werden im Anhang genannt).

Im letzten Teil des Buches möchte ich allen Anwendern einen Überblick über die bereits existierende Grafiksoftware und zusätzlichen Hardwareerweiterungen zum Amiga geben, also CAD-Software, Malprogramme, Animationsprogramme und so weiter. Diejenigen, die sich nach Lektüre dieses Werkes noch weiter für Grafikprogrammierung interessieren, finden am Schluß des Buches eine Liste mit interessanter Literatur zu den Themen Grafik und Amiga.

Abschließend muß zum Inhalt gesagt werden, daß der Schreibstil leider oft zugunsten des Fachlichen auf der Strecke bleiben mußte. Dazu muß ich allerdings betonen, daß man besser ein fachlich korrektes Buch hat, als ein fachlich miserables Produkt, das aber dann locker zu lesen ist. Es gibt mehr als ein Einsteigerbuch, das sehr lesbar ist, dessen fachlicher Inhalt jedoch nahe bei Null liegt.

Danksagungen und Randbemerkungen

Allen, die mir geholfen haben, trotz der vielen Probleme doch noch über die Runden zu kommen, soll hiermit ein herzliches »Dankeschön!« gesagt werden.

Insbesondere danke ich folgenden Personen:

- Den Wikingern und Christoph Columbus für die Entdeckung Amerikas (denn daher kommt der Amiga schließlich!)
- Herrn Scharfenberger vom Verlag Markt&Technik dafür, daß er mir die Reise auf die europäische Entwicklerkonferenz im Dezember 1985 ermöglichte (der ich viel zu verdanken habe),
- den Mitarbeitern von Commodore Deutschland dafür, daß sie sich bemüht haben, einen guten Support für Entwickler zu ermöglichen,
- Bill Kolb vom Amiga-Entwicklerteam in Kalifornien dafür, daß er seinen geplanten Newsletter über »Software, Hardware, Drugs and Rock'n' Roll« doch nie gemacht hat (den er an einem schlechten Tag in Eastbourne beim Frühstück zu produzieren vorschlug),
- Daniel Silva für das Programm DeluxePaint, mit dem ich die Entwürfe für die Illustrationen in diesem Buch gemacht habe,
- meinem Nachbarn für das geduldige Ertragen des rasselnden Druckers und der lauten Musik, die ich zur Ablenkung nötig hatte,
- Dirk Dreyer für die Übernachtungsmöglichkeit zur Hannovermesse und für sein 3D-Programm,
- Oliver Dietz dafür, daß er unbedingt in diesem Buch erscheinen wollte (was hiermit erfüllt ist)
- Meiner Freundin Silke dafür, daß sie mich nicht wie meine frühere Freundin – Störfaktor Nummer 1 bei diesem Buch – von der Arbeit abhielt, sondern sogar dazu anspornte. Ohne ihr gutes Zureden und ihre moralische Unterstützung wäre dieses Buch nicht mehr fertig geworden!

und natürlich den vielen Amiga-Freaks für Ihre Unterstützung.

Negative »Dank«sagungen an all die, die mir die Frage »Ist dein Buch schon fertig?« so oft gestellt haben, daß ich 5000 Seiten mit den ewigen Wiederholungen dieses Satzes füllen könnte. Außerdem einen

genauso »herzlichen« Gruß an diejenigen, die mir Material oder Mithilfe versprachen, aber dann doch nie etwas für mich taten.

München, 1987,

Manfred Kohlen



Buchteil 1

Dieser Teil des Buches soll Sie SYSTEMUNABHÄNGIG mit den Grundlagen der Grafikprogrammierung vertraut machen. Man kann die in diesem Teil erlernten Kenntnisse also auch mit anderen Computern verwenden. Wir werden jedoch immer wieder auf den Amiga hinweisen, um zu zeigen, wie diese Grundlagen mit dem Amiga realisiert sind oder realisiert werden können. Wer meint, bereits alles notwendige Wissen zum Thema Computergrafiken zu haben oder keine Lust auf trockene Mathematik hat, sollte zum nächsten Buchteil weiterblättern.



1 Einführung

Computergrafik ist ein Thema, das in der Computerbranche zunehmend an Bedeutung gewinnt. Grafiken waren schon immer etwas, was im Zusammenhang mit Computern die meiste Aufmerksamkeit auf sich zog – und das auch nicht zu Unrecht. Schließlich ist bekannt, daß das menschliche Auge Grafiken wesentlich schneller und besser aufnimmt als ein Zahlengewirr oder Texte. Deswegen ist die Computergrafik auch ein besonders effektives Medium zur Kommunikation zwischen Mensch und Computer. Das veranlaßte auch die meisten Computerhersteller, in ihre neuesten Geräte sogenannte »grafische Benutzeroberflächen« zu realisieren. Angefangen hat es mit größeren Systemen von Xerox. Doch richtige Verbreitung fand diese Technologie erst mit Computern wie dem Apple Macintosh, dem Atari ST und nicht zuletzt auch dem Commodore Amiga.

1.1 Wozu eigentlich Computergrafik?

Grafiken sind jedoch nicht nur für Benutzeroberflächen interessant. Eine der wohl populärsten Erfindungen unseres Jahrhunderts ist das Videospiel – das ohne Computergrafik wohl nie möglich gewesen wäre. Aber nicht nur im Heimbereich sind Grafiken notwendig. Künstler arbeiten mit Computern, um Bilder zu malen. Einige Computermagazine haben inzwischen sogar spezielle Grafik-Seiten eingerichtet, um zu demonstrieren, daß man mit dem Computer auch kreativ sein kann. Viel dieser Kreativität sieht man schon tagtäglich im Fernsehen, ohne überhaupt zu bemerken, daß man es

mit computergenerierten Grafiken zu tun hat. Bekanntestes Beispiel hierfür dürfte wohl die große 1 der ARD sein.

Reine Bilder, die ohne Bewegung auf dem Bildschirm, einem Drucker oder einem Plotter erscheinen, nennt man *statische* Computergrafiken. Manchmal sagt man auch *passive* oder (im englischen Fachjargon) *noninteraktive* Computergrafik dazu, da der Anwender keinen Einfluß auf diese Grafiken hat. Geben wir dem Anwender Kontrolle über das Bild, indem wir ihm ein Eingabegerät geben, sei es nun die Maus, ein Joystick, ein Grafiktablett oder ein Lightpen, durch das er seine Änderungen an den Computer weiterleiten kann, haben wir es mit interaktiver Computergrafik zu tun. Interaktive Computergrafiken bedingen also eine wechselseitige Kommunikation zwischen Anwender und Computer. Der Computer ändert entsprechend den Signalen, die er durch das Eingabegerät erhält, das angezeigte Bild. Der Benutzer gibt eine Serie von Kommandos an den Computer, die dieser dann verarbeitet und den entsprechenden grafischen Output erzeugt. Auf diese Weise ergibt sich ein sogenannter »Dialog« zwischen Eingabe und Ausgabe beziehungsweise zwischen Anwender und Computer.

Interaktive Computergrafik wirkt sich inzwischen sogar auf das normale gesellschaftliche Leben aus – wenn auch oft nur indirekt. So werden beispielsweise die Piloten der modernen Düsenjets in Flugsimulatoren trainiert, um unsere Sicherheit zu garantieren. Die Grafiken in einem solchen Simulator erscheinen bei heutiger Technologie mit einer Geschwindigkeit von etwa 30 Bildern pro Sekunde. Das kann der Amiga auch; das Programm »Jet« von Sublogic arbeitet mit etwa derselben Geschwindigkeit. Allerdings sind Grafiken, wie man sie in professionellen Simulatoren sieht, wesentlich komplexer als die des »Jet«. Trotzdem ist die Leistung des Amiga in dieser Hinsicht fast unglaublich.

Die Elektronikindustrie ist heute noch weitaus abhängiger von der Computergrafik als die Flugzeugindustrie. Ein typischer *Integrierter Schaltkreis*, wie er in Computern zu Dutzenden verwendet wird, ist so komplex, daß ein Entwickler eines solchen Chips etwa die zehnfache Zeit benötigen würde, um das zu zeichnen, das er jetzt mittels Computergrafik erstellt. Noch dazu kann er den Computer benutzen, um das Design zu überprüfen und notwendige Änderungen innerhalb von Minuten einzufügen. Auch der Hobbyelektroniker kann Gebrauch von der Computergrafik machen, um zum Beispiel Platinenlayouts

mit Hilfe des Computers zu entwerfen. Ein solches Programm ist übrigens auch für den Amiga schon erhältlich.

Computergrafik wird aber auch von Architekten benutzt, um Designprobleme schneller und effektiver zu lösen. Molekularforscher können Grafiken von Molekülen erstellen, um mehr Einsicht in ihre Struktur zu bekommen. (An der TH Darmstadt wird der Amiga zusammen mit großen Grafiksystemen auf diesem Gebiet eingesetzt.)

Und im Heimbereich schließlich ist die Computergrafik ein Mittel zur Kreativität (Bilder malen aus Spaß an der Freude) und zur Entspannung (Videospiele).

Warum also Computergrafik? Der Hauptgrund für den Heimanwender ist ganz klar der Spaß daran. Der wichtigste Grund für den professionellen Anwender ist zweifellos die Geschwindigkeit und Effektivität, mit der ein Computer arbeitet, und nicht zuletzt auch die wesentlich bessere Lesbarkeit eines grafischen Outputs gegenüber langen Zahlenkolonnen. Und für viele ist es einfach nur die Faszination eines neuen Mediums, die Computergrafik so interessant macht.

1.2 Geschichte der Computergrafik

Was man heute bereits im Heimcomputerbereich als selbstverständlich ansieht, benötigte lange Jahre der Forschung und Entwicklung. Schon 1950 wurde das erste computergesteuerte Display benutzt, um einfache Bilder darzustellen. Das Display war angeschlossen an den »Whirlwind I«-Computer des MIT (Massachusetts Institute of Technology). Dieses Display arbeitete mit einer sogenannten *CRT* (Cathode Ray Tube), also einer Bildröhre, ähnlich denen, die in unseren Fernsehern eingebaut sind.

Während der 50er Jahre machte die interaktive Computergrafik dann nur wenig Fortschritte. Die Computer, die zur Verfügung standen, waren nicht dafür geeignet. Sie waren lediglich Hilfsmittel für mathematische Kalkulationen. Alles, was sie erzeugen konnten, waren Zahlenkolonnen; interaktive Arbeit mit dem Computer war kaum möglich. Erst Ende der 50er Jahre wurde durch die Entwicklung von Maschinen wie TX-0 und TX-2, die ebenfalls am MIT gebaut worden waren, interaktives Arbeiten mit dem Computer möglich. Von da an nahm das Interesse an der Computergrafik sehr schnell zu.

Das Ereignis, das die interaktive Computergrafik als neues Fachgebiet erst so richtig bekannt machte, war eine Publikation von Ivan E. Sutherland, der damals gerade seinen Doktortitel am MIT erhalten hatte. Seine Arbeit hatte den Titel «Sketchpad: A Man-Machine Graphical Communication System« (zu deutsch etwa: »Sketchpad: Ein System für die Mensch-Maschine-Kommunikation«). Sutherland wurde damit zum Vorreiter der modernen Computergrafik (seine heutige Firma ist übrigens einer der erfolgreichsten Hersteller von Grafik-Simulatoren). Mit seiner Arbeit wurde zugleich der Grundstein für die erst in den 80er Jahren populär gewordenen Benutzeroberflächen gelegt.

Mitte der 60er Jahre wurden dann Computergrafik-Forschungsarbeiten in größerem Rahmen durchgeführt; unter anderem am MIT, bei General Motors, den Bell Telephone Laboratories und Lockheed Aircraft. Damit hatte das goldene Zeitalter der Computergrafik begonnen. Nachdem die 60er Jahre die Zeit der Forschung und Entwicklung war, trugen die 70er Jahre die ersten Früchte dieser Arbeit. Interaktive Computergrafiken haben sich weltweit durchgesetzt und wurden wie selbstverständlich in den verschiedensten Industriezweigen benutzt.

Bis heute hat diese Entwicklung aber nichts grundsätzlich Neues im Hinblick auf Grafiktechniken gebracht. Die grundsätzliche Funktionsweise der Computergrafiken ist in den letzten 5 Jahren gleichgeblieben; es wurden lediglich kleine Verbesserungen vorgenommen und die Technik perfektioniert. 1977 hielt die Computergrafik mit dem Apple II dann auch Einzug in den Heimbereich – und dies zu einer Zeit, in der sie im Business-Bereich noch wenig beachtet wurde. Mit dem Macintosh hielt die Computergrafik später dann auch im Bereich der »seriösen« Personalcomputer Einzug. Die Anwendungsgebiete des Business- und des Heimcomputers verschmolzen immer mehr; Computer wurden mehr und mehr mit Fähigkeiten ausgestattet, die für beide Anwendungsbereiche gleichermaßen geeignet sind. Den derzeitigen Höhepunkt in dieser Entwicklung stellt der Amiga dar. Er bietet professionelle Grafikfähigkeiten, die man bisher nur an speziellen (und sehr teuren) Grafikcomputern bestaunen konnte, in einem Preisbereich, der auch für den Durchschnittsbürger erschwinglich ist.

2 Grundwissen zur Computergrafik

Da ich in diesem ersten Buchteil vom »Punkt Null« ausgehen will, soll das folgende Kapitel einige grundsätzliche Fragen für den Anfänger beantworten. Fortgeschrittene, denen das alles schon bekannt vorkommt, können ruhig zum nächsten Kapitel weiterblättern.

2.1 Der Bildschirm

Wie ein CRT-Bildschirm (Röhrenbildschirm, wie er in heutigen Computermonitoren und Fernsehern benutzt wird) grundsätzlich funktioniert, sehen Sie in Abbildung 2.1. Am schmalen Ende der geschlossenen Röhre befindet sich eine Kathode, die einen feinen Strahl von Elektronen aussendet. Das andere Ende der Röhre besteht aus einer mit Phosphor beschichteten Fläche, dem Bildschirm. Sobald der Elektronenstrahl auf den Phosphor trifft, leuchtet dieser am entsprechenden Punkt auf. Die Energie des Elektronenstrahls kann kontrolliert werden, um die Helligkeit dieses Punktes zu steuern, oder, wenn nötig, den Elektronenstrahl so schwach zu machen, daß an dieser Stelle überhaupt kein Lichtpunkt erscheint. An der Außenseite der Röhre sind Spulen angebracht, die durch Anlegen verschiedener Spannungen verschieden starke elektromagnetische Felder erzeugen. Die elektromagnetischen Felder lenken den Elektronenstrahl ab. Der Computer macht also nichts anderes, als durch seine Videologik entsprechende Spannungen an die Magneten des Monitors abzugeben.

Die Lichtintensität des Phosphors fällt jedoch schnell, sobald der Strahl vorbeigerauscht ist. Um ein ganzes Bild auf dem Bildschirm erscheinen zu lassen, muß der Elektronenstrahl also immer wieder über die Stellen fahren, die aufleuchten sollen. Diesen Vorgang nennt man »Refresh-Prozeß«. Damit das Bild nicht flimmert, muß er mindestens 30mal pro Sekunde durchgeführt werden.

Jeder Fernseher und auch RGB-Monitor arbeitet im wesentlichen genau mit dieser Technik. Bei Farbbildschirmen besteht ein Bildpunkt allerdings immer aus drei einzelnen Phosphorpunkten, einem Roten, einem Grünen und einem Blauen. Durch die Mischung der drei Farben mit verschiedenen Intensitäten erscheint dieser Punkt dem menschlichen Auge, als hätte er eine bestimmte Farbe. Wie das funktioniert, sehen Sie in Bild 2.2. Der Amiga ist speziell für solche (RGB-) Bildschirme entwickelt worden.

Neben dem CRT-Bildschirm gibt es auch noch die DVST (Direct View Storage Tube), eine Abwandlung des üblichen CRTs, bei dem einmal mit dem Elektronenstrahl angestrahlte Punkte leuchten bleiben, die LCD-Displays, die mit Flüssigkristallen arbeiten, sowie die Plasmabildschirme.

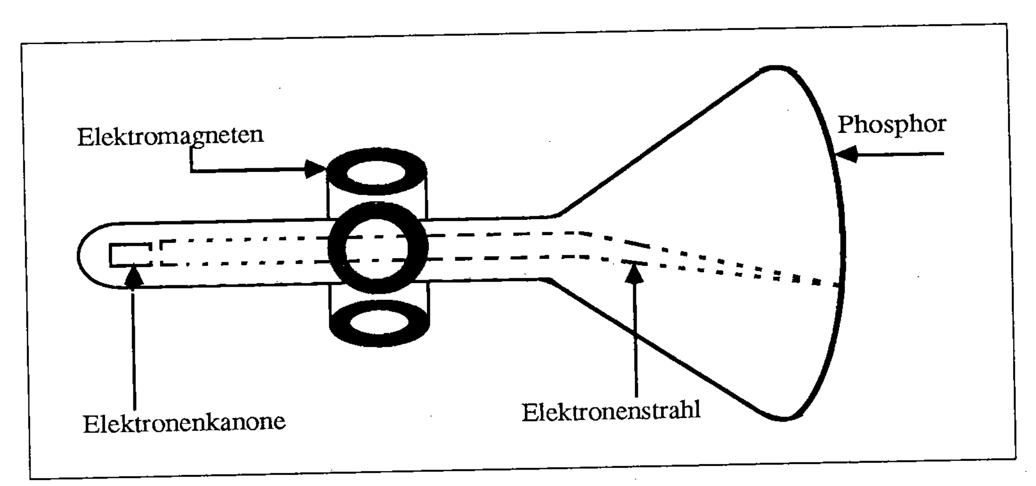


Bild 2.1: Der Grundaufbau eines CRT-(Röhren-)Bildschirms

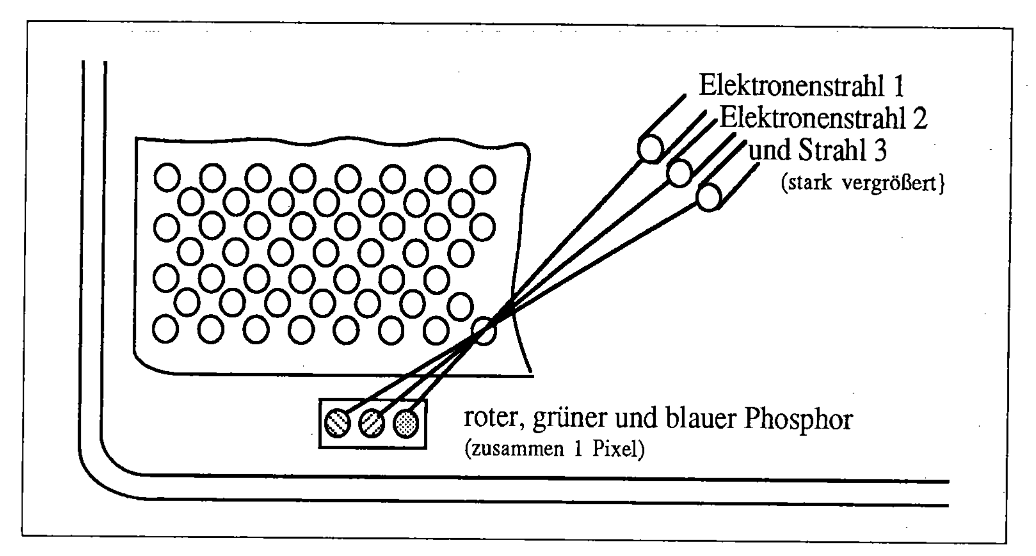


Bild 2.2: Funktionsweise der Lochmaske eines Röhrenbildschirms

2.2 Display-Techniken

Computergrafiken kann man grundsätzlich in zwei Klassen einteilen: Pixelgrafiken und Vektorgrafiken. Beide Arten von Computergrafiken erscheinen nicht nur unterschiedlich auf dem Bildschirm, sondern benötigen auch völlig unterschiedliche Techniken zu ihrer Erzeugung. Vektorgrafiken, auch »Line Drawings« genannt, sind wesentlich einfacher zu erzeugen, und der für sie benötigte Speicherplatz ist geringer als bei Pixelgrafiken. Pixelgrafiken, gibt es erst seit der zweiten Hälfte der 60er Jahre, als die sogenannten »Frame Buffer Displays« entwickelt wurden (dazu später noch mehr).

Mit Vektorgrafiken lassen sich nur Linien darstellen. Dabei wird durch entsprechende Schaltungen der Elektronenstrahl der Bildröhre direkt zwischen bestimmten Punkten hin- und her bewegt, um die benötigten Linien zu erzeugen. Bei der Pixelgrafik wird dagegen der Elektronenstrahl Zeile für Zeile über den Bildschirm gelenkt. Dabei wird aus dem Speicher, in dem die Grafiken liegen, dem »Frame Buffer«, die Information geholt, ob der Punkt, auf dem der Rasterstrahl sich gerade befindet, gesetzt ist oder nicht. Jeden Bildpunkt nennt man ein »Pixel«, was eine Verballhornung von »Picture Element« (Bild-Element) ist.

Bis Ende der 70er Jahre war die Vektorgrafik die meistbenutzte Technik in der professionellen Computergrafik-Industrie. Moderne CAD-Anwendungen benötigen nämlich typischerweise größere und aufwendigere Grafiken als Heimanwendungen. Da die Vektorgrafik speicherplatzsparender ist als die Pixelgrafik und Computerspeicher (RAMs) bis vor kurzem noch sehr teuer waren, hatte die Vektorgrafik klare Preisvorteile. Inzwischen sind jedoch die RAMs billiger geworden. Daraufhin hat die Pixelgrafik an Bedeutung gewonnen und ist jetzt, Mitte der 80er Jahre, zur Standardtechnik geworden. Auch der Amiga benutzt Pixelgrafik. Wie diese Grafik speziell beim Amiga aufgebaut ist, erfahren Sie in Buchteil 2.

Bevor der Amiga auf dem Markt kam, gingen Gerüchte um, er würde neben der Pixelgrafik auch noch Vektorgrafik unterstützen. Diese Gerüchte waren jedoch unbegründet und sind auf die Tatsache zurückzuführen, daß der Amiga Hardware besitzt, die Linien (Vektoren) extrem schnell zeichnen kann. Diese Linien werden aber mit der normalen Pixelgrafik-Methode auf den Bildschirm gebracht.

2.3 Das Frame-Buffer-Prinzip

Ein modernes Grafikdisplay besteht aus drei wichtigen Komponenten: Einem digitalen Speicher, dem »Frame Buffer«, in dem das Bild gespeichert ist, einem Bildschirm und einem einfachen Interface (Schnittstelle) zwischen den beiden, dem »Display Controller«. Im Frame Buffer wird die Grafik als Matrix von binären Werten abgelegt, also beispielsweise ein gesetztes Bit (1) für einen leuchtenden Punkt und ein nicht gesetztes Bit (0) für ein schwarzes Pixel. Wie das genau funktioniert, sehen Sie in Abbildung 2.3. Der Display Controller ist die Schaltung, die diese Umsetzung zwischen Speicher und Bildschirm vornimmt und die Bitgruppierungen des Frame Buffers in Signale umwandelt, die der Monitor verarbeiten kann.

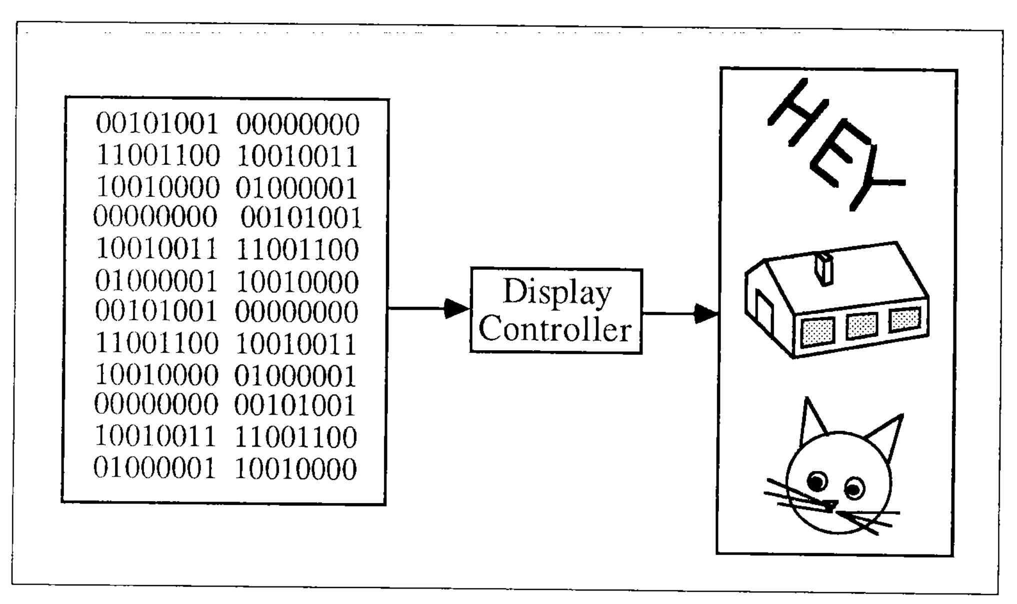


Bild 2.3: Das Prinzip des Frame Buffers



3 Grundlagen der Grafikprogrammierung

3.1 Punkte, Linien und andere geometrische Grundlagen

Was ist ein Punkt? Geometrisch gesehen ist ein Punkt eine Position in einem Raum. Ein Raum hat eine unendliche Anzahl von möglichen Punkten. Jede Position in einem Raum hat also ihren eigenen dazugehörigen Punkt. Ein Punkt hat keine Größe, er ist infinitesimal klein.

In einem Bild hat ein »Punkt« allerdings eine bestimmte Größe. Wir können den Punkt sogar (durch Vergrößerung mit einer Lupe) als einen Kreis oder ein Rechteck sehen. Würden wir jedoch einen geometrischen Punkt vergrößern wollen, geht das nicht. Ein Punkt bleibt immer ein Punkt (ohne Ausdehnung), egal welche Vergrößerung wir benutzen.

Nehmen wir jetzt zwei Punkte A und B. Zwei Punkte haben zwei Positionen in einem Raum. Haben beide Punkte dieselbe Position, haben wir es mit nur einem einzigen Punkt zu tun, dem wir jedoch zwei verschiedene Namen gegeben haben. Sind die Positionen nicht gleich, besteht eine Distanz zwischen beiden. Verbinden wir zwei Punkte, haben wir eine Linie zwischen A und B. Linien besitzen immer zwei Endpunkte und bestehen aus allen Punkten, die auf der gradlinigen Verbindung zwischen diesen beiden Endpunkten liegen. Da Punkte unendlich klein sind, ist eine Linie also eine Ansammlung von unendlich vielen Punkten.

Was ist nun eine Fläche? Linien haben nur Länge, sie sind eindimensional. Flächen haben zwei Dimensionen, Länge und Breite. Wir können das geometrische Konzept einer Fläche als ein unendlich dünnes Blatt Papier sehen, das sich in Länge und Breite ausdehnt. Eine Fläche wird im Englischen auch »Plane« genannt. Daher stammt auch der Ausdruck »Bitplane«, den Sie im Zusammenhang mit dem Amiga oft hören. Eine Bitplane ist eine »Bitfläche«, also eine Ansammlung von Bits auf einer Fläche.

Uns interessiert nun, wie wir einen Punkt in eine Fläche setzen (färben), denn Punkte setzen ist die grundlegende (einfachste) Operation der Computergrafik. Wäre die Grafikebene eindimensional, bräuchten wir einen Punkt auf dieser Linie nur als eine einzige Zahl spezifizieren. Erinnern Sie sich dazu bitte an den guten alten Zahlenstrahl aus der Schule: Der Mittelpunkt dieses Strahls ist die Position 0, auch Ursprung genannt. Wir können jeden Punkt auf dem Strahl durch eine Zahl ausdrücken, indem wir einfach die Entfernung des Punktes vom Ursprung nennen, wobei Punkte links vom Ursprung negative Entfernungen besitzen.

Eine Fläche, also auch ein Bildschirm oder ein Blatt Papier, ist aber zweidimensional. Hier können wir den Punkt nur durch zwei Zahlen beschreiben. Dazu benutzen wir ein sogenanntes »Koordinatensystem«. Vom Ursprung aus geht hier ein Zahlenstrahl in horizontale Richtung nach rechts und einer in vertikale Richtung nach oben. Der eine Zahlenstrahl wird »x-Achse«, der andere wird »y-Achse« genannt. Wir geben die Position des Punktes also durch zwei Zahlen, eine »x-Koordinate« und eine »y-Koordinate«, an. Hierzu ein Beispiel: Der Punkt (5,8) ist 5 Einheiten rechts und 8 Einheiten oberhalb vom Ursprung zu finden. Diese Einheiten sind in der Geometrie abstrakte Zahlen, in der Praxis jedoch meist Zentimeter oder andere Maßeinheiten für Entfernungen. Auf dem Computer sind diese Einheiten die Pixels, also die Punkte, aus denen sich das Bild zusammensetzt.

Ein solches Koordinatensystem erlaubt es uns also, Punkte auf einer Fläche mit Hilfe von Zahlen darzustellen. (Ohne Zahlen geht bei Computern sowieso so gut wie gar nichts.) Ich sagte weiter oben, eine Fläche würde aus unendlich vielen Punkten bestehen. Geometrisch gesehen stimmt das auch. Doch ein Computer kann nicht unendlich viele Punkte auf einer endlichen Fläche darstellen. Im Computer verwenden wir daher ein Koordinatensystem mit einer endlichen

Anzahl von Einheiten (Pixels). Die Anzahl der auf einem Computer möglichen Punkte nennt man die »Grafikauflösung« des Computers.

Koordinaten für einzelne Punkte innerhalb dieser Auflösung können am Computer nicht mit Brüchen (rationalen oder reellen Zahlen) dargestellt werden, weil es nur ganze Bildpunkte gibt. (Haben sie schon mal einen halben Bildpunkt gesehen? Halbe Pixels gibt es nicht!). Wird also irgendwo in einem Programm mit nicht ganzzahligen Werten für Bildschirmkoordinaten gearbeitet, so wird dies durch Software (in der Programmiersprache oder im Betriebssystem) oder Hardware in ganze Zahlen umgewandelt. In manchen Programmiersprachen muß dies noch vom Benutzer selbst gemacht werden. (99% der Computer ignorieren dabei einfach die Nachkommastellen.)

Wir wissen jetzt also, wie wir dem Computer sagen, welche Position auf dem Bildschirm (oder Drucker bzw. Plotter) er setzen (färben) soll.

Wie teilen wir dem Computer nun mit, wie er eine Linie ziehen soll? Wir erinnern uns: Eine Linie muß aus mindestens zwei (End-) Punkten bestehen und verläuft in einer bestimmten Richtung. Die Richtung einer Linie in einem zweidimensionalen Koordinatensystem (also einer Fläche) nennt man ihre »Steigung«. Die genaue Position einer Linie wird zudem noch durch den »y-Achsenabstand« (den Punkt, an dem die Linie die y-Achse schneidet) festgelegt. Die Gleichung einer Geraden lautet $y = mx+b$, wobei m die Steigung und b der y-Achsenabschnitt ist.

Eine mathematische Gerade mit unendlicher Länge auf dem Computer darzustellen, ist aber wieder einmal nicht möglich, weil wir keinen unendlich großen Bildschirm und auch keinen unendlich großen Framebuffer haben. Wir müssen uns bei der Computergrafik also mit Strecken statt Geraden begnügen. Wir betrachten also nur die Punkte, die zwischen zwei Endpunkten $p1$ und $p2$ liegen. Um eine Linie auf dem Computer zu setzen, genügt es nicht, die beiden Enden anzugeben (ausgenommen bei Vektorgrafikgeräten). Wir müssen stattdessen alle Punkte setzen, die auf der Linie zwischen $p1$ und $p2$ liegen (also die Bedingung erfüllen, die die entsprechende Geraden-Funktion stellt). Wie wir so etwas berechnen, erfahren Sie auf den folgenden Seiten. Der Amiga besitzt übrigens Hardware, die selbständig Linien ziehen kann und dem Programmierer die Berechnung

der Koordinaten der Punkte zwischen den beiden Endpunkten abnimmt.

Um Computergrafiken erstellen zu können, sollte man das Prinzip des Vektors ebenfalls verstehen. Anders als Geraden und Strecken haben Vektoren keine feste Position im Raum. Sie haben lediglich eine bestimmte Länge und Richtung. Einen Vektor können wir beschreiben durch die Entfernung des Vektorendpunktes vom Vektorursprung in horizontaler und vertikaler Richtung, also durch relative x- und y-Koordinaten. Vektoren sind insbesondere notwendig für die Berechnung von Dehnungen und Rotationen von geometrischen Figuren.

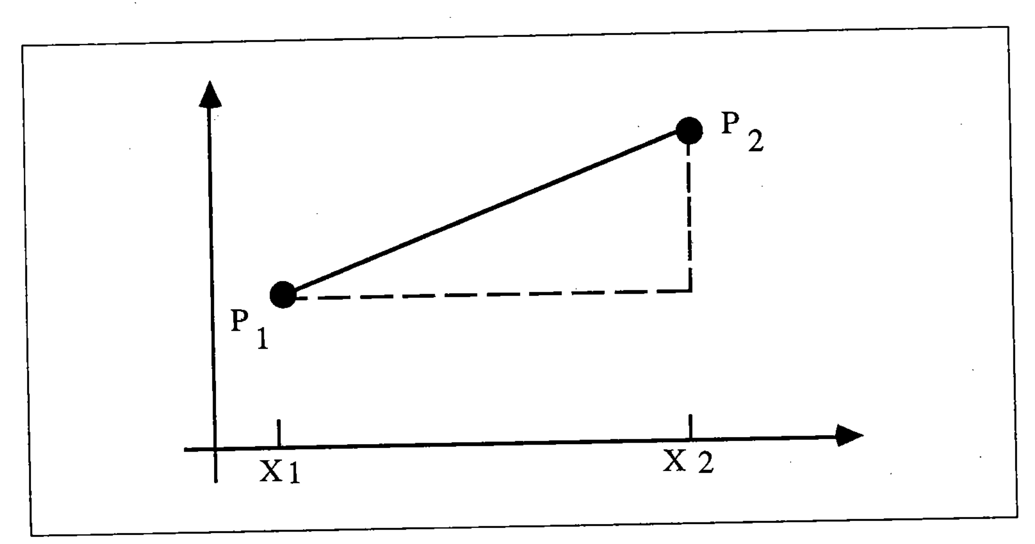


Bild 3.1: Ein Line-Segment

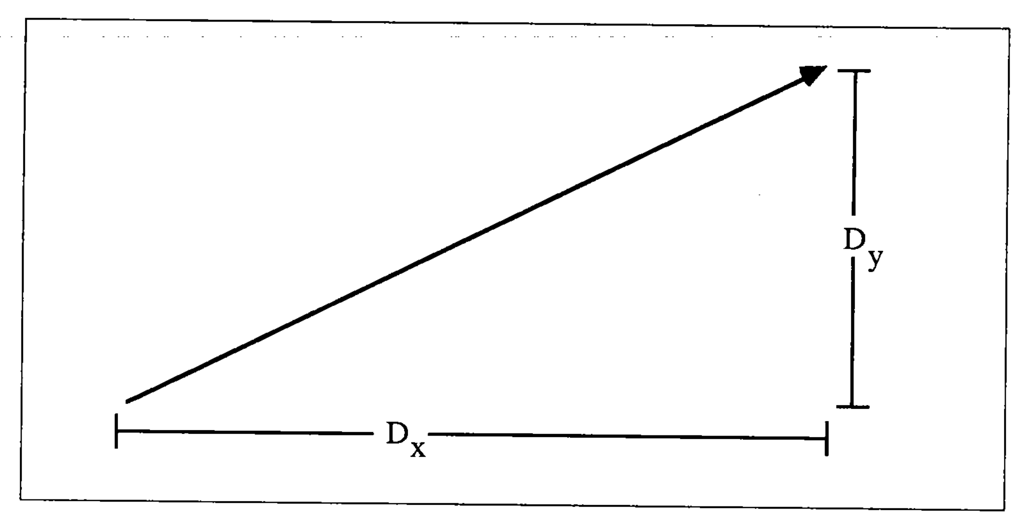


Bild 3.2: Ein Vektor

3.2 Umsetzung der geometrischen Grundlagen

Wie vereinbart sich das bisher Gelernte nun mit einem Grafikdisplay eines Computers? Wie bereits gesagt, können wir keine unendlich große Anzahl von Punkten auf den Bildschirm bringen, wir müssen mit einer begrenzten Anzahl von Pixels arbeiten. Diese begrenzte Anzahl wird Auflösung genannt und wird meistens in Länge mal Breite angegeben (zum Beispiel 320 mal 200 Pixels). Die Auflösung eines Computers heißt auf englisch *resolution*. Hat ein Computer verschiedene Grafikauflösungen, so werden sie unterschiedlich benannt: *Hires* (*high resolution*) für die höherauflösende Grafik und *lores* (*low resolution*) für die gering auflösende Grafik. Beim Amiga wird beispielsweise die Grafik mit 320 Punkten Breite als *lores* bezeichnet, die Grafik mit 640 Punkten als *hires* (die verschiedenen Auflösungen in vertikaler Richtung sind nicht als verschieden große Auflösungen im herkömmlichen Sinn zu sehen. Vergleichen Sie dazu auch später die Beschreibung des *Interlace-Modus* im Buchteil 2).

Jeder Punkt eines Bildes ist mit seinen Koordinaten adressierbar. Ein Pixel ist das kleinste adressierbare Element eines Computergrafik-Displays. Jedes Pixel entspricht bei Schwarzweißgrafiken einem Bit im Speicher. Bei Farbgrafiken werden für die Farbkodierung mehrere Bits als Farbinformation für ein einzelnes Pixel benutzt.

Eine Matrix von Bits ist eine Bitplane. Der Speicher, der diese Bitplane enthält, ist der Frame Buffer. Wollen wir eine Linie malen, müssen wir also die einzelnen Bits des Frame Buffers zwischen dem Anfangspunkt *p1* und dem Endpunkt *p2* setzen, und zwar so, daß die Pixelkoordinaten der Linien-Bits die Bedingung erfüllen, die durch die mathematische Funktion der Gerade (von der die Linie ein Teil ist) erfüllen. Wollen wir einen Kreis, ein Rechteck oder eine andere geometrische Figur malen, so müssen ebenfalls die einzelnen Punkte dafür berechnet und gesetzt werden.

Die folgenden Seiten sollen Ihnen zeigen, wie man so etwas – als Grundlage für spätere Programmierarbeiten – berechnet. In Computergrafik liegt viel Mathematik, doch ohne die kommen wir nun mal nicht aus. Sie werden jetzt also von Algorithmen und Formeln förmlich überfallen.

Alle diese Algorithmen sollen jedoch nur zeigen, wie es funktioniert. Wir beziehen uns daher nur auf Punktekoordinaten, und nicht auf Speicheradressen (die der Programmierer noch benötigen würde, um die entsprechenden Bits im Frame Buffer setzen zu können).

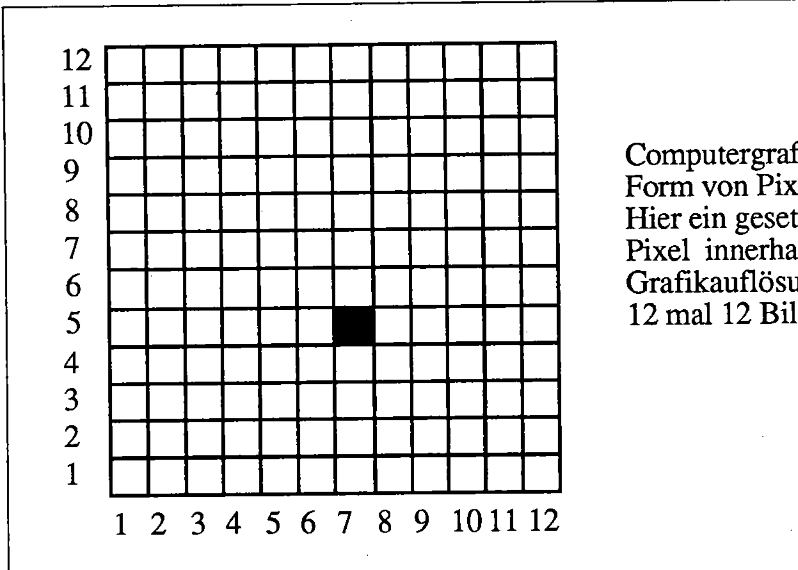


Bild 3.3: Was ist ein Pixel?

3.3 Erste Algorithmen: Linien ziehen

Der Prozeß des Linienziehens wird im Fachjargon »Vektor Generation« genannt. Das kommt daher, daß wir Linien ziehen, indem wir vom Startpunkt aus eine relative Bewegung in der Länge eines Vektors machen. Die Methode des Linienziehens, die die meistgebräuchliche ist, nennt sich »einfache symmetrische digitale Differenzanalyse«, im (englischen) Fachausdruck auch »simple symmetrical digital differential analyzer (DDA)«. Die Methode wird im folgenden kurz »DDA« genannt.

Wie funktioniert nun der DDA-Algorithmus? Dazu müssen wir erst eine weitere mathematische Ausdrucksweise für eine Geradengleichung kennen; wir nennen diese Form die »parametrische« Form, weil wir die x- und y-Koordinaten in Form eines Parameters *u* angeben. Nehmen wir an, wir wollen eine Linie zwischen den Punkten (x1,y1) und (x2,y2) ziehen. Wir sagten, daß wir jeden Punkt zwischen diesen beiden Koordinaten so setzen wollen, daß dabei eine gerade Linie herauskommt. Das heißt, wir müssen die x-Koordinate in konstanten Schritten von *x1* nach *x2* wandern lassen.

Das können wir mit der folgenden Gleichung ausdrücken:

$$x = x1 + (x2 - x1)\ u$$

Wenn u Null ist, ist *x* gleich *x1*. Hat *u* den Wert 1, bewegt sich *x* dabei zum Punkt *x2*. Für eine Linie, die nicht nur waagrecht von links nach rechts gehen soll, sondern auch schräg nach oben oder unten, müssen wir **gleichzeitig** die y-Koordinate von *y1* nach *y2* bewegen. Dabei gilt die folgende Gleichung:

$$y = y1 + (y2 - y1)\ u$$

Beide Gleichungen zusammen beschreiben eine gerade Linie. Diese Gleichungen liefern uns den richtigen Ansatz zur Berechnung der Pixels, die wir zwischen Anfangs- und Endpunkt setzen müssen. Wir starten dabei einfach mit $u = 0$ und inkrementieren *u* in kleinen Schritten, bis wir bei $u = 1$ angelangt sind. Während wir dies tun, bewegen sich die x- und y-Koordinaten in kleinen Schritten entlang der Linie. Und das ist genau das, was wir für das Zeichnen einer Linie benötigen: Wir zählen einfach *u* um den besagten kleinen Schritt

hoch, berechnen dann mit dem neuen *u*-Wert die x- und y-Koordinate des neuen Punktes und setzen diesen Punkt.

Wie gehen wir das jetzt von einem Programm aus an? Der erste Teil des Algorithmus ~~besteht daraus, zu~~ berechnen, wie oft wir *u* hochzählen müssen, also wieviel Schritte wir benötigen. Wir beginnen damit, zu bestimmen, wie -- weit wir in x- und y-Richtung malen müssen. Dazu generieren wir den Vektor (dx,dy), den wir aus den horizontalen und vertikalen Abständen zwischen Anfangs- und Endpunkt der Linie berechnen. Diesen Vektor müssen wir in genügenden Arbeitsschritten aufteilen, so daß kein Schritt größer wird als ein Pixel. Werden nämlich zu wenig Schritte genommen, erscheinen Lücken in der Linie, weil Pixels »übersprungen« werden. Wie das aussieht, sehen Sie in Abbildung 3.4a. Da die Länge der Linie in Pixels gegeben ist, ist die minimale Anzahl der Schritte, die wir benötigen, um eine Linie ohne Unterbrechungen zu zeichnen, die absolute Länge der größeren Vektorkomponente (entweder die x- oder die y-Komponente). Das heißt, daß wir zum Beispiel den x-Wert des Vektors als Schrittanzahl benützen, sofern der Abstand zwischen Anfangs- und Endpunkt in x-Richtung größer ist, als der Abstand zwischen den y-Werten. Andernfalls (wenn der y-Abstand größer ist als der x-Abstand) verwenden wir den y-Wert des Vektors als Anzahl der Schritte. Natürlich können wir auch mehr Schritte machen als diese Mindestanzahl. Das führt aber dazu, daß die Linie dicker wird oder das ein und dasselbe Pixel mehrmals gesetzt wird. Letzteres beeinträchtigt die Effektivität des Algorithmus in bezug auf Geschwindigkeit. Gewiß, es ist das kleinere von zwei Übeln, doch die beste Methode ist es eben doch, genau die richtige Anzahl von Schritten zu nehmen. Bild 3.4b und 3.4c zeigen, wie die richtige und eine größere Anzahl von Schritten auf dem Bildschirm wirken.

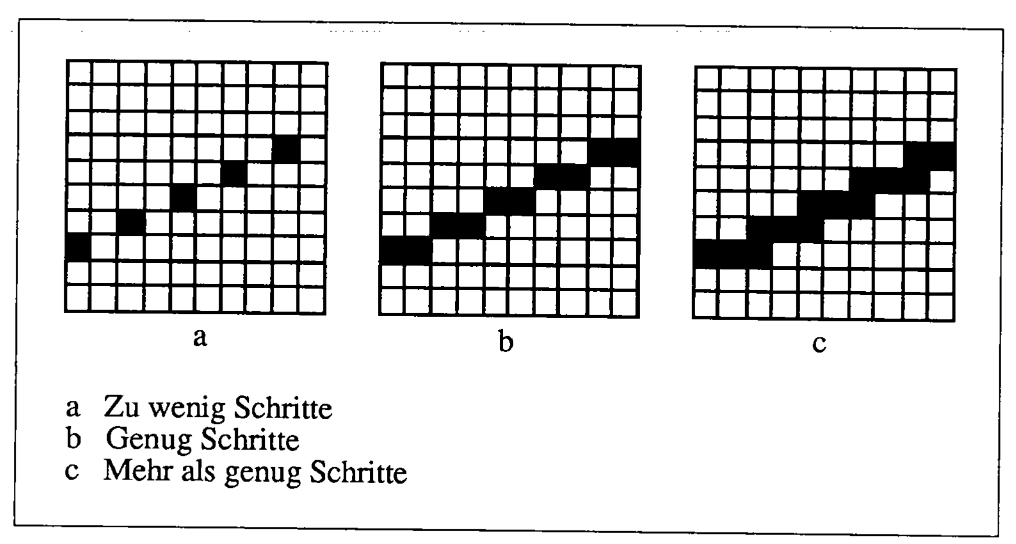


Bild 3.4: Effekte bei Ändern des Line-Algorithmus

Eine Schrittweite gleich einem Pixel ist eigentlich optimal. Wir werden jedoch eine Schrittweite von 2 verwenden. Der Grund dafür ist, daß der Versuch, eine Schrittweite von genau einem Pixel zu verwenden, durch Rundungsfehler zu einer etwas größeren Schrittweite führt. Und das kann bekanntlich zu Unterbrechungen der Linie führen (siehe Bild 3.4).

Nachdem wir die Anzahl der Schritte berechnet haben, müssen wir die Schrittweiten in x- und y-Richtung ermitteln, um die die Position jeweils erhöht werden soll. In unserem Programmablaufplan (Abbildung 3.5), der den Algorithmus verdeutlichen soll, sind diese Schrittweiten in den Variablen *xs* und *ys* enthalten. Wir erhalten diese Werte, indem wir die Vektorlängen durch die Anzahl der Schritte (steps) teilen. Durch diese Methode benötigen wir die eingangs genannten Gleichungen, die die Grundlage zu all dem hier beschriebenen bilden, nicht mehr.

Würden wir die Gleichungen benutzen, kämen wir jedoch auch auf die obige Methode: Wir könnten jeweils ein bißchen zu u hinzurechnen und dann x und y durch die beiden Gleichungen berechnen. Wir bemerken, daß x und y sich beide um einen bestimmten Betrag erhöht haben. Wir sehen dabei jedoch auch, daß sich dabei x und y immer um denselben Betrag erhöhen, wenn wir u um denselben Betrag erhöhen. Wenn wir das erst mal richtig erkannt haben, brauchen wir x

und *y* jeweils nur noch um dieselben konstanten Werte zu erhöhen (*xs* und *ys*); der Parameter *u* wird dadurch unwichtig. Wir benützen in unserem Beispiel (siehe Programmablaufplan) das *u* lediglich noch als Laufvariable für das Mitzählen der bereits vollzogenen Schritte.

Bevor wir nun die Schleife zum Setzen der Punkte angehen können, haben wir noch ein Problem zu lösen: Wir wissen, daß es nur ganze Pixels gibt und wir aus diesem Grunde keine Kommazahlen verwenden dürfen. Nun kommt es aber vor (und das sehr oft!), daß eine geometrisch exakte Linie zwischen zwei Pixels liegen würde – beispielsweise bei einer y-Koordinate von 0.48. Setzen wir nun das Pixel darüber oder darunter? Unser Computer macht aus Kommazahlen ganze Zahlen, indem er einfach die Nachkommastellen abschneidet (= Integerberechnung). Das Problem liegt darin, daß bei einer solchen Berechnung der Koordinaten alle Punkte unter der geometrisch exakten Linie liegen würden. Eine bessere Lösung ist die Rundung. Eine Rundung können wir auch erzeugen, indem wir zum jeweiligen Wert 0.5 hinzuzählen und dann erst die Nachkommastellen entfernen. Machen wir das mit den Anfangskoordinaten, mit denen die Zählschleife beginnt, dann brauchen wir nur ganz zu Anfang diese 0.5 Punkte hinzuzuzählen. Diese Addition gilt dann für die gesamte Schleife, da wir innerhalb der Schleife von den geänderten Anfangswerten ausgehen. Somit ist die x-Koordinate des ersten zu plottenden Punktes der Integer-Wert von *x1* plus 0.5 (in BASIC: *x = x1+0.5*). Das gleiche gilt für den y-Wert. Durch diese Rundung garantieren wir, daß wir die Punkte, die der geometrischen Linie am nähesten sind, setzen.

Schließlich und endlich kommt die Schleife, die einfach die Laufvariable hochzählt, die Änderungen der x- und y-Werte um die entsprechende Schrittweite berechnet, den Punkt setzt, und bei Erreichen der vorher berechneten Schrittanzahl endet. Zur Schleife muß allerdings gesagt werden, daß durch Schleifenbefehle in bestimmten Programmiersprachen nur aufwärts gezählt werden kann und somit nur Linien von links nach rechts gezogen werden können. Um das zu vermeiden, kann man eine Hilfsvariable einführen, die den Wert -1 annimmt, wenn die Linie von rechts nach links gehen soll. Wie das in BASIC funktioniert, sehen Sie in einem der Kapitel über Grafikprogrammierung in BASIC.

Natürlich gibt es außer dem eben beschriebenen einfachen DDA noch weitere Linienalgorithmen, so zum Beispiel den Bresenham-Algo-

rithmus, der sich durch einige Vereinfachungen mit reiner Integer-Arithmetik (also ohne Nachkommastellen) und nur mit Additionen (bei völliger Vermeidung von Multiplikation und Division) benutzen läßt. Der Bresenham-Algorithmus eignet sich daher hervorragend für die Programmierung in Maschinensprache und für die Implementation in Hardwarechips. (Mit welchem Algorithmus der Chip des Amiga seine Linien zieht, ist mir bisher nicht bekannt.) Wer zu weiteren Algorithmen mehr Informationen haben will, sollte sich das Literaturverzeichnis am Ende des Buches betrachten.

Da der Amiga bereits selbst Linien ziehen kann, benötigen Sie das Wissen über diese Algorithmen zur Programmierung Ihres Amiga eigentlich nicht. Sie werden hier nur als Grundwissen zum Thema Computergrafik erklärt! Für jemanden, der seine Amiga-Programme auf andere Systeme umschreiben will, wird sich dieses Wissen allerdings vielleicht als sinnvoll erweisen.

dx = x2 - x1 — Vektor x-Länge

dy = y2 - y1 — Vektor y-Länge

dx > dy ? (ja: steps = dx; nein: steps = dy) — Anzahl der nötigen Schritte festlegen

steps = steps + 1 — Um 1 erhöhen wegen eventueller Rundungsfehler

xs = dx/steps
ys = dy/steps — Schrittweite pro u-Inkrement berechnen

x = x1 + 0.5
y = y1 + 0,5 — Anfangskoordinaten festlegen (inklusive Rundungskorrektur)

u = u + 1 — Schleifenzähler

u > steps ? (ja: end) — notwendige Schritte-Anzahl ⟹ Linie beendet

nein:

Punkt setzen (auf Integerwert von x/y) — nähesten an geometrischer Gerade liegenden Punkt setzen

x = x + xs
y = y + ys — zum nächsten Punkt

Bild 3.5: Ablauf eines Line-Algorithmus

3.4 Textdarstellungen in Grafikdisplays

Auch Text gehört zur grafischen Gestaltung. Schließlich wollen wir zu technischen Zeichnungen entsprechende fachliche Erläuterungen geben oder in einer Businessgrafik Verkaufszahlen und Marktanteile mit den entsprechenden Beträgen und Beschriftungen verdeutlichen. Unser Problem als Programmierer liegt nun wieder darin, das Ganze auf dem Bildschirm oder einem anderen Ausgabegerät darzustellen. Auch dazu gibt es wieder verschiedene Techniken. Dabei unterscheiden wir zwischen zwei grundsätzlichen Methoden: Die »Stroke«-Methode und die »Dot-Matrix«-Methode.

Die Stroke-Methode wird hauptsächlich für Vektorgrafikdisplays und Plotter verwendet. Ein Buchstabe besteht dabei aus Linien, die auf den Bildschirm gemalt werden. Für jeden Buchstaben ist eine bestimmte Linienfolge vorgesehen. Der Buchstabe A besteht beispielsweise aus drei Linien. Der Vorteil dieser Methode besteht darin, daß wir einfach nur die Länge der Linien mit zwei multiplizieren müssen, wenn wir doppelt so große Schrift erhalten wollen. Damit ist es möglich, problemlos Schriften in beliebiger Größe darzustellen.

Methode zwei ist die »Dot Matrix«-Methode. Wir benützen dabei eine Punktematrix, in der der Buchstabe abgelegt ist. Eine solche Matrix ist praktisch ein sehr kleines Bild, in dem der entsprechende Buchstabe als Anordnung von Pixeln dargestellt wird. Diese Matrix wird zur Darstellung der Schrift einfach in den Frame Buffer kopiert. Viele Computer verwenden dazu Hardware, den sogenannten Character-Generator-Chip – man kann es aber auch (langsamer) in Software machen. Die meisten dieser Character-Generator-Chips sind jedoch begrenzt, was heißen soll, daß jeder Buchstabe genau eine bestimmte Größe haben muß (beispielsweise beim Commodore 64 in eine 8 * 8 Matrix passen muß). Neben diesen festen Matrizen gibt es auch noch die Proportionalschrift-Technik. Die Matrixbreite ist dabei variabel und hängt von dem jeweiligen Buchstaben ab. Damit können wir bei kleineren Buchstaben entsprechend mehr Text in einer Zeile unterbringen. Der Vorteil von Proportionalschrift ist eine bessere Übersicht, besonders in grafikfähigen Textverarbeitungssystemen mit Blocksatzgenerierung. (Auf dem Bildschirm ist schon besser zu erkennen, wie der Ausdruck aussehen wird.) Diese Methode muß meist durch Software, und kann

nicht in Hardware realisiert werden. Auch der Amiga benutzt diese Technik. Die Buchstabenmatrizen und die Routinen zur Textausgabe sind in der Betriebssystemsoftware enthalten.

Wenn wir die Größe des Textes mit der Dot-Matrix-Methode verändern wollen, benötigt das allerdings eine wesentlich kompliziertere und langsamere Methode als bei Stroke-Schriften. Wir müssen zur Vergrößerung und Verkleinerung die gesamte Buchstabenmatrix vergrößern, also jeden einzelnen Punkt neu berechnen. Aus diesem Grunde wird für schnelle Grafikprogramme oft die Stroke-Methode verwendet, obwohl der Computer in seiner Hardware oder Betriebssystem-Software die Dot-Matrix-Methode implementiert hat. Am Amiga haben wir das nicht nötig, wie man an den schnellen Vergrößerungen und Verkleinerungen im Malprogramm »Deluxe Paint« sieht, das als Standard für professionelle Amiga-Grafiksoftware gesehen werden kann. Soll das ganze Projekt aber nicht in einer Compilersprache (wie Turbo Pascal), sondern mit einem Interpreter (wie Microsoft Amiga-BASIC) durchgeführt werden, wird der Vergrößerungs- und Verkleinerungsalgorithmus extrem langsam. Die Algorithmen zu Dehnungen und Streckungen von Pixelgrafiken sind zudem auch zu kompliziert für Anfänger.

Am Amiga sind die verschiedenen Textmatrizen bereits in unterschiedlichen Größen auf der Workbench-Diskette gespeichert. Hier muß also für die unterschiedlichen Textgrößen nichts berechnet werden. Als Nachteil bleibt aber bestehen, daß wir an die festen Größen, wie sie auf der Diskette gespeichert sind, gebunden sind.

Interessant für die Textgestaltung können auch verschiedene Schriftarten sein. Mit der Matrix-Methode werden hierzu einfach weitere Matrizen von Diskette geladen (oder sind bereits im Speicher). Schriftänderungen wie Fettdruck, Kursivschrift und ähnliches kann man aber bereits durch kleine Berechnungen erreichen, ohne gleich neue Zeichensätze laden zu müssen. In der Regel sind diese Änderungsberechnungen schneller als das Nachladen von Diskette.

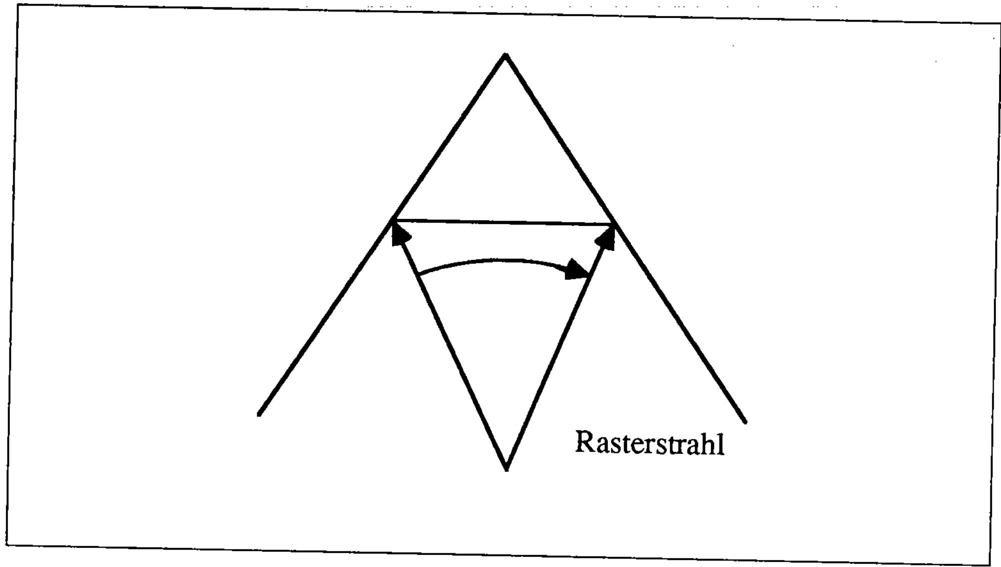


Bild 3.6: Die Stroke-Methode

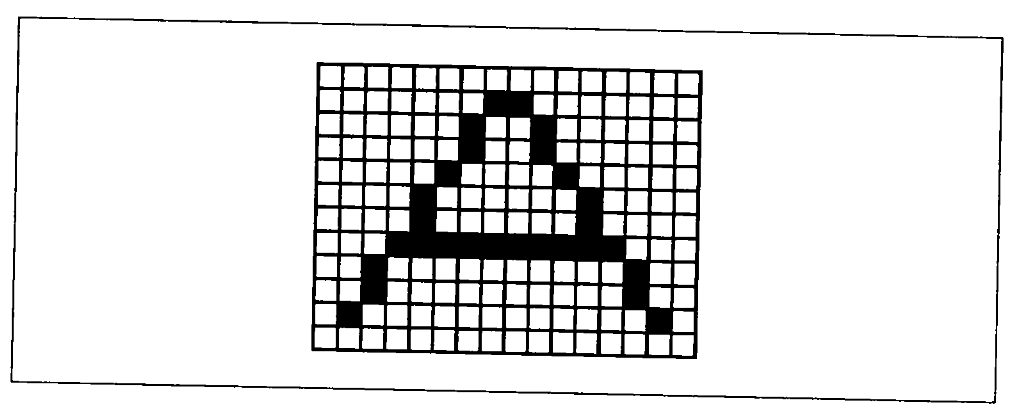


Bild 3.7: Die Matrix-Methode

Solche Routinen sind ebenfalls in den Grafikroutinen des Amiga-Betriebssystems enthalten. Wie das in BASIC funktioniert, zeigt das auf jeder Extras-Diskette mitgelieferte Beispiel »Fonts«. Für Leute, die entsprechendes später auf andere Computer umsetzen wollen, soll hier kurz erläutert, wie man solche Schriftänderungen generiert: Zur Erstellung eines fettgedruckten Buchstabens wird der gesamte Buchstabe einfach zweimal ausgedruckt, das zweite Mal jedoch um ein Pixel nach rechts verschoben. So erscheint ein »dicker« Buchstabe.

Schräggestellte Buchstaben (Italics) werden berechnet, indem wir die oberste Zeile der Zeichenmatrix um einen bestimmten Wert nach rechts verschieben, die nächste Zeile um einen geringeren Betrag verschieben und so weiter, bis wir bei der letzten Zeile angekommen sind, die überhaupt nicht verschoben wird. Wie die Ergebnisse aussehen, sehen Sie in Bild 3.8.

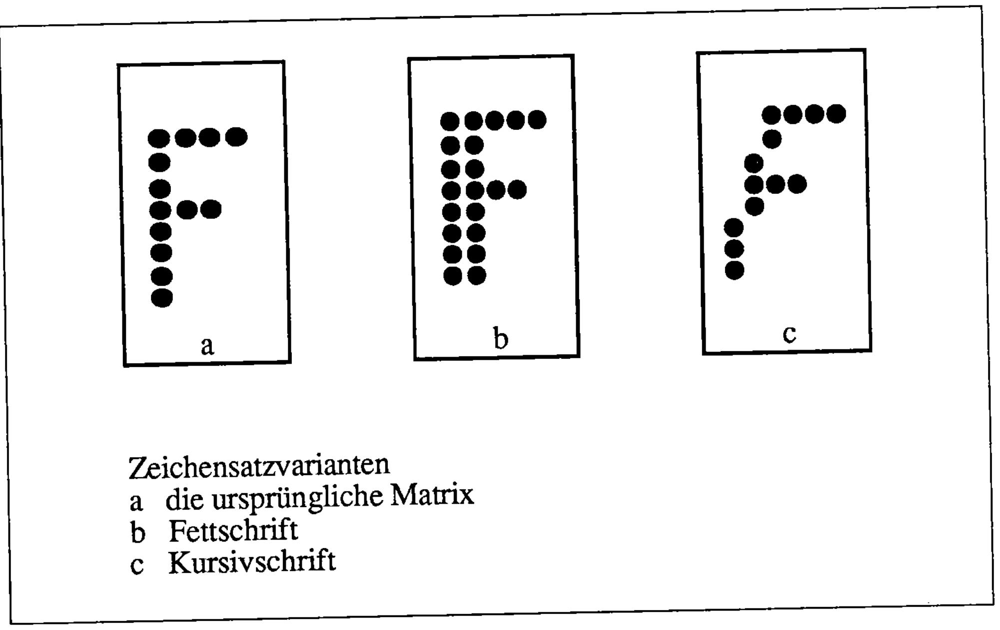


Bild 3.8: Durch einfache Manipulation erzeugte Schriftarten

3.5 Kreise und Bögen

Kreise und Bögen sind speziell bei CAD-Programmen von großer Bedeutung. Für die Programmierung von einfachen grafischen Darstellungen in Businessgrafiken braucht man sie nicht unbedingt, doch sie verschönern den optischen Eindruck beachtlich. Für Spiele ist die Erzeugung von Kreisen und Bögen zu langsam, dort werden sie, wenn benötigt, zumeist vorher berechnet und in einem Teil des Speichers abgelegt, von dem aus sie sehr schnell in den Frame Buffer übertragen werden können.

Vektorgrafikdisplays können zumeist keine echten Kreise erzeugen, sondern setzen Kreise aus Liniensegmenten zusammen. Je weniger Liniensegmente, desto schneller die Grafik, dafür aber sieht der Kreis auch um so eckiger aus. Je mehr Liniensegmente, um so kreisähnlicher wird das Bild. Eine Ansammlung von unendlich vielen Liniensegmenten, bei denen die Steigung jeder Linie minimal von der der vorherigen Linie abweicht, ergibt einen Kreis. Praktisch gesehen ist ein Kreis also eine Ansammlung von Tangentenpunkten vieler Geraden, deren Steigung sich jeweils um eine kleine Einheit *e* ändert (siehe Bild 3.9).

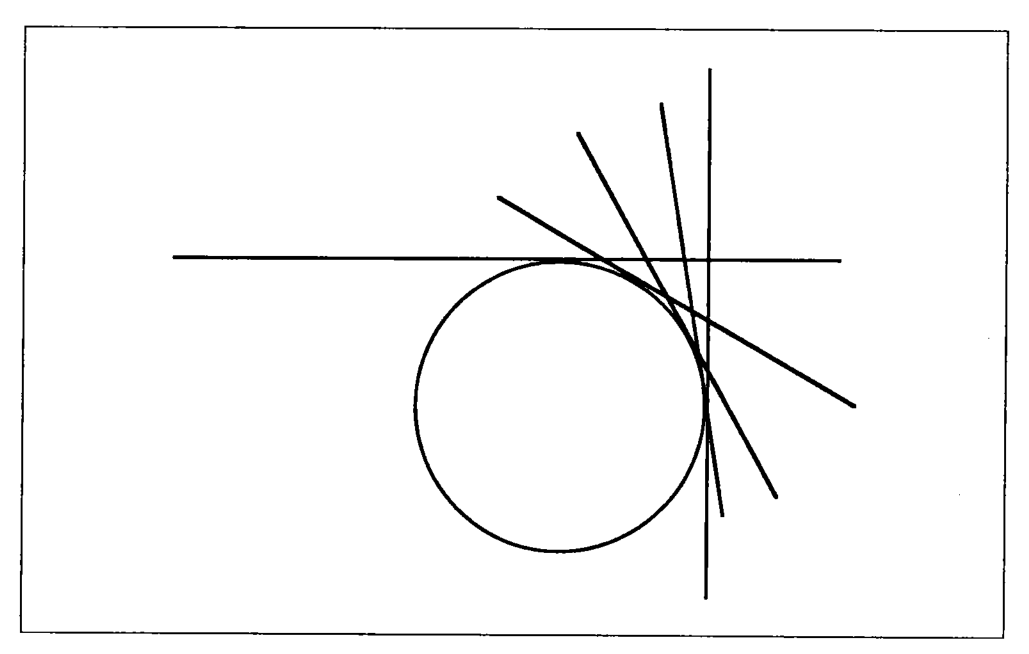


Bild 3.9: Berechnung von Kreisen durch Steigungen von Tangenten

Aus diesem Grunde können wir das Prinzip des oben beschriebenen DDA auch auf andere als gerade Linien anwenden: Kreise, Ellipsen, Spiralen und alle anderen gebogenen Linienformen – man muß nur jeweils die Einheit *e* zur Errechnung der Ableitung (Steigung der Tangente an einem Punkt der Kurve) entsprechend pro Schritt ändern.

Die »Differentialgleichung« eines Kreises mit dem Zentrum im Ursprung des Koordinatensystems lautet:

$$\frac{dy}{dx} = \frac{-x}{y}$$

Damit können wir einen kreiszeichnenden DDA entwickeln, indem wir *-ex* und *ey* als Schrittweiten benutzen werden (wobei *e* eine kleine Zahl ist, die die Anzahl der Schritte pro Einheit/Pixel angibt).

Wir kommen damit auf folgende Gleichungen, um von Punkt (x_n, y_n) auf den nachfolgenden Punkt (x_{n+1}, y_{n+1}) zu gelangen:

$$x_{n+1} = x_n + e\, y_n$$

$$y_{n+1} = y_n - e\, x_n$$

Dabei müssen wir bei jedem Schritt die Schrittweiten erneut berechnen, denn wie bereits oben erwähnt, muß für jeden nachfolgenden Punkt auch die Steigung geändert werden. Diese Berechnung können wir allerdings auf eine Stellenverschiebung und eine Komplementbildung reduzieren, wenn *e* als negative Potenz von 2 gewählt wird. Um dem Aufeinanderfolgen von Punkten über eine Bildschirmeinheit hinaus vorzubeugen, sollte e gleich 2^{-n} sein. Dabei gilt:

$$2^{n-1} \leq r \leq 2^n$$

Dabei ist r der Kreisradius. Die gerade beschriebene Methode bildet allerdings eine Spirale statt eines Kreisbogens. Das passiert, weil jeder Schritt in einer Richtung parallel zum Kreisradius gemacht wird; daher ist jeder Punkt ein bißchen weiter vom Mittelpunkt entfernt als der Punkt zuvor. Wenn wir jedoch x_{n+1} statt x_n zur Berechnung der nachfolgenden y-Koordinate y_{n+1} benutzen, können wir dieses Problem umgehen. Dadurch ergibt sich:

$$x_{n+1} = x_n + e\, y_n$$

$$y_{n+1} = y_n - e\, x_{n+1}$$

Diese Lösung basiert auf dem Gedanken, daß wir die Gleichung des DDA (zweite genannte Gleichung dieses Kapitels) auch als Matrixform schreiben können:

$$[x_{n+1}\ \ y_{n+1}] = [x_n\ \ y_n] * \begin{bmatrix} 1 & -e \\ e & 1 \end{bmatrix}$$

Wenn Sie noch keine Matrizenrechnung kennen, empfehle ich als Einführung ein gutes Mathematikbuch (Empfehlungen siehe Bibliographie im Anhang dieses Buches).

Die Determinante auf der rechten Seite entspricht nicht einer Einheitsmatrix, sondern $1+e^2$. Um die Kurve nicht mehr spiralförmig, sondern als geschlossenen Kreis darstellen zu können, reduzieren wir die Determinante auf eine Einheitsmatrix:

$$[x_{n+1}\ y_{n+1}] = [x_n\ y_n] * \begin{bmatrix} 1 & -e \\ e & 1-e^2 \end{bmatrix}$$

Führen wir diese Matrixgleichung wieder auf eine »normale« Gleichung zurück, erhalten wir die zuletzt genannte Gleichung, die x_{n+1} zur Berechnung von y_{n+1} heranzieht.

Kreise, die mit diesem DDA erzeugt werden, müssen nicht unbedingt das Zentrum im Nullpunkt haben. Stattdessen wird die Verschiebung in x und y des Kreismittelpunktes (x_n,y_n) gebraucht, um (x_{n+1},y_{n+1}) zu bestimmen. Wir ändern also nur die Ursprungskoordinaten, der Rest der Kreisberechnung baut auf diese Anfangskoordinaten auf. Daher ist dieser Algorithmus auch gut für eine Hardware-Implementation geeignet – vielleicht tut dies das nächste oder übernächste Gerät der Amiga-Familie. Die Modelle 500, 1000 und 2000 besitzen entsprechende Chips leider noch nicht.

Da für diese Methode auch nur Additionen und Subtraktionen notwendig sind, läßt sie sich auch leicht in Maschinensprache realisieren. Das Problem an diesem einfachen Kreis-DDA ist, daß wir es zwar mit einer geschlossenen Kurve zu tun haben, aber wir wieder keinen exakten Kreis erreichen. Wir haben es vielmehr mit einer Ellipse zu tun. Bei einer großen Schrittweite *e* wird das sehr deutlich. Je kleiner *e* wird, desto kreisähnlicher wird die Ellipse. Aber je kleiner *e* wird, desto mehr Schritte werden auch benötigt, um den Kreis zu vervollständigen. Dadurch dauert die Berechnung deutlich länger. Einen exakten Kreis erhalten wir, wenn wir mit Sinus- und Cosinus-Funktionen arbeiten. Dadurch ersparen wir uns auch unnötig viele Rechenschritte. Je nach Computer ist die eine oder andere Methode schneller. (Die Berechnung trigonometrischer Funktionen dauert meist recht lange.) Amiga-BASIC benutzt übrigens die DDA-Methode ohne trigonometrische Funktionen, um mit ein und demselben Algorithmus auch problemlos Ellipsen darstellen zu können.

Vielleicht waren Ihnen diese mathematischen Ausführungen zu kompliziert. Für diesen Fall wollen wir das Problem noch einmal, aber von einer anderen Seite her, angehen. Diesmal ändern wir nicht die Schrittweite für die Berechnung der Steigung von Geraden (deren Tangentenpunkte dann den Kreis bilden), sondern gleich den Winkel. Dadurch können wir auch Teilkreise zeichnen, also beispielsweise einen Viertelkreis oder einen Achtelkreis. Gut anwendbar ist diese Möglichkeit für sogenannte »Pie-Charts« oder »Tortengrafiken«, wie sie auch bei den Wahlhochrechnungen verwendet werden.

Die Gleichungen, um die Koordinaten eines Punktes zu finden, der bei einem bestimmten Winkel auf dem Rand eines Kreises liegt, lauten:

$$x = r \cos A + x_0$$

$$y = r \sin A + y_0$$

Wobei x und y die beiden Koordinaten des gesuchten Punktes sind, x_0 und y_0 den Mittelpunkt des Kreises darstellen, A den Winkel im Bogenmaß angibt und r natürlich der Radius ist. Differenzieren bringt uns zum nächsten Punkt des Kreises. Hierzu werden x um dx und y um dy geändert, wenn sich der Winkel um dA ändern. Diese Werte errechnen sich nach der Differenzierung wie folgt:

$$dx = -r \sin A \, dA$$

$$dy = r \cos A \, dA$$

Ändern wir nun Gleichung 1, indem wir x_0 und y_0 auf die andere Seite der Gleichung bringen, ergibt sich:

$$r \cos A = x - x_0$$

$$r \sin A = y - y_0$$

so daß wir zum Schluß der gesamten Angelegenheit herausbekommen:

$$dx = -(y - y_0)\, dA$$

$$dy = (x - x_0)\, dA$$

Aus diesen Gleichungen können wir dann auch einen Algorithmus zum Erstellen von vollständigen Kreisen entwickeln. Dieser läuft im wesentlichen wie folgt ab:

Finden eines passenden Inkrement-Wertes für Winkel :

dA= (Betrag(x-x_0 + Betrag(y-y_0))*pi, oder, falls dieser Wert größer als 0.01 ist, 0.01 (zumeist reicht 0.01 aus).

Ersten Punkt setzen

Solange gemalter Winkel noch nicht gleich dem Endwinkel des zu malenden Teilkreises ist:

neue x-Koordinate = x-Koordinate + (y_0-y-Koordinate)*dA

neue y-Koordinate = y-Koordinate + (x-Koordinate - x_0)*dA

neuer Winkel = vorher gemalter Winkel + dA

Punkt (x/y) setzen

zurück an Anfang für nächsten Punkt (solange gemalter Winkel...)

Anhand dieser Skizze könnten Sie nun schon einen einfachen Kreis-Algorithmus in BASIC entwickeln (wenn sie unbedingt auf den CIRCLE-Befehl von BASIC verzichten wollen). Sie können diesen Algorithmus aber auch verwenden, um eine kreisförmige Bewegung eines Objektes wie zum Beispiel eines Sprites zu berechnen. Natürlich können Sie zu ähnlichen Zwecken auch den Line-Algorithmus »mißbrauchen«, um eine geradlinige Bewegung eines Objektes von einem Punkt zum anderen zu berechnen. Genau das tun wir später auch im Kapitel »Animationstechniken«. Und zwei der Gleichungen, aus denen wir die Koordinaten eines Punktes auf dem Kreisrand berechneten, werden im 3D-Programm im Beispielekapitel des Basicteils verwendet, um die Drehung des 3D-Objektes zu berechnen. Sie sehen also, daß diese mathematischen »Ausschweifungen« durchaus einen Sinn haben.

Vielleicht wird es Sie geärgert haben, daß die Winkelangaben in allen bislang erwähnten Formeln im Bogenmaß erfolgen müssen. Das Bogenmaß geht von 0 bis π (PI), während Sie (und ich) üblicherweise mit Gradangaben zwischen 0 und 360 arbeiten. Diese Angaben lassen sich jedoch mit sehr einfachen Mitteln umrechnen. Wenn Sie in der Schule entsprechend aufgepaßt haben, wissen Sie es ja bereits:

Bogenmaß = Winkelmaß * 0.017444444444

Eine mögliche Anwendung hierfür möchte ich nun auch gleich skizzieren: Mit dem oben genannten Algorithmus und der Umrechnung von Winkel- in Bogenmaß können Sie nun schon Kuchengrafiken

zeichnen, wie sie oft zur Veranschaulichung von Umsätzen verwendet werden. Dazu malen Sie die entsprechenden Teilkreise. Verbindungen der Teilkreise mit der Mitte (um farbig ausgefüllte Kuchenstückchen darstellen zu können) nehmen Sie dann mit Hilfe einer Linie vor. Dazu müssen Sie den Mittelpunkt des Kreises mit dem Anfangs- und Endpunkt des Teilkreises verbinden. Beispiel: Sie wollen eine Wahlbeteiligung von 25 Prozent darstellen. Dazu brauchen Sie einen Teilkreis von 0 bis 90 Grad oder von 90 bis 180 und so weiter. Den Mittelpunkt kennen Sie bereits. Woher aber die Endpunkte für die zwei Linien nehmen? Ganz einfach: Wir verwenden wieder die Gleichung 1 unserer zweiten Kreismethode:

$x = r \cos A + x_0$

$y = r \sin A + y_0$

Also errechnen Sie den Endpunkt eines Kreises mit Winkel 90 und Mittelpunkt (50/50) bei einem Radius von 10 mit:

$x = 10*\cos(90*0.0174444)+x_0$

$y = 10*\sin(90*0.0174444)+y_0$

So würden auch die entsprechenden Zeilen eines Basicprogramms aussehen. Diese Angaben können Sie auch zur Vervollständigung des mit Absicht fehlenden Textes im Basicteil verwenden. Mal sehen, ob Sie dieses Problem lösen können ...

Neben diesen noch recht einfachen Kreisgeneratoren gibt es auch weitere Möglichkeiten, gebogene Formen wie die besagten Kreise, Ellipsen, Spiralen oder andere Kurven darzustellen. Die wichtigste Technik dazu nennt sich auch »Spline-Verfahren« und beruht auf Interpolation. Der Spline-Algorithmus berechnet eine komplette Kurve anhand von einzelnen Punkten der Kurve. Damit können dann auch Parabeln oder Exponentialfunktionen gezeichnet werden, und auch »gebogene Linie« wie in DELUXE-PAINT sind kein Problem. Diesem Thema wollen wir uns allerdings nicht mehr widmen, denn es ist viel zu kompliziert für ein Einsteiger-Kapitel. Dies hier ist nun mal ein Buch für Anfänger und soll es auch bleiben.

3.6 Sonstige zweidimensionale geometrische Objekte

Die Grundformen der geometrischen Objekte und deren Umsetzung durch Algorithmen auf den Computer haben wir nun bereits durchgesprochen. Weiterhin gibt es aber auch noch Polygone, abgeschlossene Objekte mit gekrümmter Oberfläche, und dergleichen mehr.

Polygone sind nichts weiter als aneinandergesetzte Linien. Der Programmierer muß zur Erzeugung von Polygonen nur deren Eckpunkte kennen. Zwischen diesen Eckpunkten werden Linien gezogen, und schon ist das Objekt fertig. Manchmal wissen wir aber auch nur einen Mittelpunkt, um den herum ein regelmäßiges Polygon gezeichnet werden soll. Auch hier hilft wieder die Mathematik weiter. Dazu läßt sich nämlich eine Abwandlung des DDA anwenden, der eine Kombination aus dem Kreis-Algorithmus und den Linien-Algorithmen ist. Mit dem Kreis-Algorithmus berechnen wir in regelmäßigen Winkelabständen (für ein 5-seitiges Polygon zum Beispiel alle 72 Grad) die Koordinaten der Punkte auf dem Kreisrand und verbinden diese Punkte dann einfach mit Linien.

Schwieriger wird es da schon bei gekrümmten Objekten, zum Beispiel dem Querschnitt durch eine Blumenvase. Dabei können wir uns der ungefähren Form durch Linien und Teilkreise annähern, doch eine exakte symmetrische Vase werden wir kaum zustande bekommen. Hier helfen wieder die Spline-Algorithmen, die mir allerdings zu kompliziert erschienen und die wir hier auch nicht besprechen werden. Durch eine geschickte Mixtur aus Teilkreisen und Linien läßt sich aber schon ein ganz annehmbares und zumeist auch realistisches Bild erzeugen.

Damit wären wir aber auch schon so ziemlich alle Grundlagen, die Sie zur Arbeit mit zweidimensionalen Grafiken benötigen, durchgegangen. Bei der Arbeit mit einer Programmiersprache (BASIC, C, oder was auch immer) werden Sie sicherlich darauf zurückgreifen müssen – auch, wenn Sie es jetzt noch nicht glauben. Das sind aber wirklich nur die Grundlagen. Es gibt nämlich noch eine ganze Menge mehr Wissenswertes über zweidimensionale Grafiken. Da wäre beispielsweise das »Clipping«, das es ermöglicht, einen Körper an bestimmten Stellen abzuschneiden. Und mit Vergrößerungen und

Verkleinerungen wollen wir uns lieber erst gar nicht plagen, denn hier schlägt die Mathematik voll zu.

3.7 Dreidimensionale Grafiken

Dreidimensionale Grafiken üben nicht nur eine ungemein starke Faszination auf den Betrachter aus, sondern bieten auch Unmengen von interessanten und nützlichen Anwendungsmöglichkeiten. Wer kennt nicht computererzeugte Werbegrafiken, die sich scheinbar im Raum drehen; 3D-Grafik wird zur Konstruktions-Unterstützung im Maschinenbau verwendet; der Chemiker kann sich, ausgehend von einem abstrakten Modell, mit dem Computer die dreidimensionale Struktur von Molekülen verdeutlichen (wie das die TH Darmstadt nicht nur auf einem größeren System, sondern auch auf dem Amiga macht). Auch Simulationen komplexer Bewegungen oder einfach nur Trickfilme können mit dem Computer erstellt werden.

Gehen wir nun zur Veranschaulichung der grundlegenden Prinzipien der 3D-Computergrafik einmal ein Beispiel durch. Es handelt sich dabei um einen Quader im Raum. Ein Quader hat 8 Eckpunkte, die miteinander durch Linien verbunden werden müssen. Die Positionen der Punkte werden durch die drei Werte für die Koordinaten x, y, und z angegeben. Nehmen wir nun einmal an, der Quader soll verschiedene Längen, Breiten und Tiefen haben (sonst wäre er ja ein Würfel). Dazu nehmen wir uns zunächst einen Würfel vor und »skalieren« diesen. Das bedeutet, daß jede Koordinate jedes Punktes durch Multiplikation mit Skalierungsfaktoren gestaucht oder verlängert wird, je nachdem, wie wir es haben wollen. Das geschieht durch die folgenden Formeln:

$x_{fs} = x_f*sx$

$y_{fs} = y_f*sy$

$z_{fs} = z_f*sz$

Mit *fs* werden die normierten Koordinaten des ursprünglichen Objektes (des Würfels) bezeichnet, quasi die Koordinaten unseres »Figursystems«, in dem wir unsere Figur gespeichert haben. *sx*, *sy* und *sz* geben dann die Skalierung in der jeweiligen Richtung an. Durch die Skalierungsmethode können wir also bei Ändern des sx-, sy- und sz-Wertes jedes beliebige 3D-Objekt dehnen und stauchen. Der Index *sf*

bezeichnet schließlich die Koordinaten des sich daraus ergebenden Punktes im 3D-Koordinatensystem.

Nicht ganz so einfach ist es, die Drehung eines Objekts im 3D-Koordinatensystem zu berechnen. Dazu gleich aber noch mehr. Erst nach Abschluß dieser Berechnungen wird das Objekt in die zwei Dimensionen des Bildschirms projiziert (umgerechnet). Unser Quader soll zunächst in beliebiger räumlicher Lage in die 3D-Szene eingefügt werden. Die Drehung geben wir durch drei Winkel (einen für jede Achse) an, um die wir drehen wollen. Für jeden Drehwinkel erstellen wir dann eine Drehmatrix. Wie Sie aus der Schule wissen, ist eine Matrix ein rechteckiges Zahlenschema, das sich mit anderen Matrizen nach bestimmten Rechenvorschriften verknüpfen läßt. Figuren werden durch Matrizenmultiplikation mit der Rotationsmatrix räumlich verändert, der Körper dreht sich im Koordinatensystem, wenn seine Eckpunkte mit den entsprechenden Drehmatrizen multipliziert werden.

Die Drehmatrix für eine Rotation des Quaders um die x-Achse lautet (wobei *A1* den Winkel der Drehung um die x-Achse angibt):

$$D_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(A1) & -\sin(A1) \\ 0 & \sin(A1) & \cos(A1) \end{bmatrix}$$

Analog dazu lautet die Matrix für eine Drehung um den Winkel *A2* um die y-Achse:

$$D_Y = \begin{bmatrix} \cos(A2) & 0 & \sin(A2) \\ 0 & 1 & 0 \\ \sin(A2) & 0 & \cos(A2) \end{bmatrix}$$

Und nun schließlich folgt auch noch die Matrix für die Drehung um die z-Achse:

$$D_Z = \begin{bmatrix} \cos(A3) & -\sin(A3) & 0 \\ \sin(A3) & \cos(A3) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Mit diesen Formeln können wir ganz einfach jeden Eckpunkt transformieren. Wir müssen also beispielsweise bei einer Drehung um 20 Grad in x- und 30 Grad in y-Richtung die Koordinaten eines jeden Punktes zunächst mit der Matrix D_X und dann mit der Matrix D_Y multiplizieren. Insgesamt müssen wir dann 16 Transformationen durchführen (für jeden Eckpunkt zwei). Die Berechnung der neuen

Koordinaten für einen Punkt nach der Drehung um 20 Grad um die x–Achse sieht zum Beispiel wie folgt aus:

$$[x_{neu}\ y_{neu}\ z_{neu}] = [\,x\ y\ z\,] * \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(20) & -\sin(20) \\ 0 & \sin(20) & \cos(20) \end{bmatrix}$$

Wir verwenden also die Koordinaten des Punktes ebenfalls als (einzeilige) Matrix und multiplizieren sie mit D_x. Daraus errechnen sich die neuen Koordinaten des ersten Punktes nach der x-Transformation. Nun folgt die y-Transformation: Den neu errechneten Punkt multiplizieren wir mit der Matrix Dy. Damit hätten wir bereits 2 Transformationen durchgegangen. Jetzt folgen weitere zwei Transformationen für den zweiten Punkt, zwei für den dritten und so weiter, bis wir beim letzten Punkt angekommen sind. Jede Transformation mit einer Matrix entspricht – auf »normale« Mathematik zurückverfolgt – der Berechnung von drei Gleichungen. Für eine Drehung eines Quaders in x- und y-Richtung müssen wir also 48 Gleichungen berechnen.

Haben wir die Drehung berechnet, wollen wir das Objekt vielleicht noch im Raum verschieben. Die Verschiebung vollzieht sich dabei für jeden Punkt nach der Rechenvorschrift:

$$x_{(verschoben)} = x_{(berechnete\ Koordinate)} + x_0$$

$$y_{(verschoben)} = y_{(berechnete\ Koordinate)} + y_0$$

$$z_{(verschoben)} = z_{(berechnete\ Koordinate)} + z_0$$

wobei x_0, y_0 und z_0 die Verschiebung auf der jeweiligen Achse darstellen. Soll das Objekt näher an die x-Achse herangeholt werden, müssen wir natürlich einen negativen Wert für x_0 angeben, der uns sagt, um wieviele Punkte unseres Koordinatensystems sich das Objekt zur x-Achse hinbewegt.

Damit ist die Berechnung der Objekte im dreidimensionalen Koordinatensystem abgeschlossen. Wie Sie an den Berechnungen der letzten drei Seiten sehen, können wir mit ein paar wenigen Gleichungen Drehungen um jede Achse mit der Drehmatrix vornehmen, Veränderungen des Objektes selbst durch Skalierung mit den Skalierungsfaktoren *sx* bis *sz* berechnen und schließlich auch noch die Lage im Raum durch einfache Addition oder Subtraktion (wie eben beschrieben) verändern.

Das hier ist der Grundstein zur Berechnung dreidimensionaler Objekte. Wir könnten nun auch noch mit Algorithmen beginnen, die verdeckte Flächen auch wirklich verdecken, doch das soll uns vorerst nicht interessieren; es würde viel Mathematik und etwa 50 Seiten an weiteren Voraussetzungen erfordern. Auch ohne »Hidden Lines« können wir schon ganz akzeptable 3D-Grafiken erzeugen. Zur Darstellung dieser Grafiken müssen wir die Koordinaten eines jeden Punktes aber noch in ein 2D-System übertragen – der Bildschirm ist schließlich platt!

3.7.1 Verschiedene dreidimensionale Objekte

Bevor wir unser Beispiel vom Quader fortführen und seine Eckpunkte in Bildkoordinaten umsetzen, möchte ich Ihnen aber noch kurz verdeutlichen, wie Sie andere Objekte – zum Beispiel eine Kugel – durch 3D-Koordinaten darstellen können. Eine Kugel können Sie natürlich schlecht durch Angabe von Eckpunkten darstellen. Eine Kugel hat nun mal keine Ecken. Was also tun?

Ganz einfach: Wir beschränken uns auf »Nicht-Ganz-Kugeln«. Eine Kugel stellen wir durch eine Kombination verschiedener Teile dar. Stellen Sie sich einen 20seitigen regelmäßigen Körper (Polyeder) vor. Der sieht schon fast (!) aus wie eine Kugel. Uns soll das vorerst genügen. (Wie Sie die Eckpunkte eines solchen Körpers berechnen, können Sie in jedem Geometriebuch nachlesen.) Wir nähern uns also der echten Form des Objektes nur an. Eine Möglichkeit dieser Annäherung an das Originalobjekt wäre bei der Kugel die Berechnung einzelner Punkte durch Radius, Winkelbereiche für Meridian und Achsrotation. Die bei der Berechnung herausgekommenen Werte können dann ebenso für die Transformationen im 3D-Raum verwendet werden wie die des Quaders. Grundsätzlich gilt: An jede Figur kann man sich durch gezielte mathematische Berechnungen zumindest annähern. Körper mit gekrümmten Oberflächen werden durch Polyeder annähernd echt dargestellt.

3.7.2 Umrechnung dreidimensionaler Objekte in Bildkoordinaten

Die Darstellung einer 3D-Grafik ist auf einem 2D-Bildschirm natürlich nicht möglich. Wir müssen daher die 3D-Koordinaten erst

einmal in Bildschirmkoordinaten umrechnen. Dabei gilt es, die räumliche Tiefe des Bildes optimal und treffend darzustellen.

Das können wir durch ein ganz einfaches Gedankenmodell realisieren: Wir stellen uns vor, das Bild wird auf eine Leinwand projiziert. Dazu brauchen wir zwei Dinge: Ein Projektionszentrum und eine Leinwand. Die Leinwand ist logischerweise der Bildschirm. Das Projektionszentrum können wir uns als die Stelle vorstellen, an der bei einem Diaprojektor die Glühbirne sitzt.

Um die 2D-Bildkoordinaten zu bestimmen, wird also eine Projektion des 3D-Objektes auf die Leinwand vorgenommen. Wir verfolgen dazu den Lichtstrahl vom Projektionszentrum durch einen Punkt des projizierten Objektes bis zur Leinwand. Natürlich können wir nicht für alle Punkte auf dem Bildschirm diese Berechnung vornehmen. Wir beschränken uns nur auf die Lichtstrahlen, die die Eckpunkte unseres Objektes schneiden. Die Leinwand, also unser Bildschirm, wird in der Fachsprache auch »Bildebene« genannt. Da wir, wie gesagt, nur die Lichtstrahlen berücksichtigen wollen, die durch die Eckpunkte unseres 3D-Objektes gehen, berechnen wir Geraden von der Projektionsebene bis hin zur Bildebene.

Alle Projektionsstrahlen sind dann parallel und schneiden die Bildebene im rechten Winkel. Diese Abbildungsmethode ist sehr einfach zu realisieren: Wir bilden das Objekt auf einer Ebene des Koordinatensystems ab, indem wir eine Komponente der Eckpunktkoordinaten gleich Null setzen. Dabei ist eine lineare Zuordnung zwischen dem abzubildenden Rechteck des Objektkoordinatensystems (3D-Realsystem) und dem Rechteck auf dem Bildschirm vorzunehmen. Diese lineare Rechtecktransformation würde allerdings nur die Seitenansicht eines Objektes ohne perspektivische Darstellung liefern. Sie verdeutlicht aber ganz gut die Funktionsweise der Transformation von Objektkoordinaten zu Bildkoordinaten.

Schneiden sich alle Projektionsstrahlen in einem Punkt, dem Projektionszentrum (»Glühbirne«), haben wir es mit einer perspektivischen Abbildung oder »Zentralprojektion« zu tun. Das würde auch in etwa der Arbeitsweise unserer Augen entsprechen. Für eine solche Zentralprojektion müssen wir Abbildungsparameter bestimmen, wie wir sie vom Diaprojektor oder auch von der Fotografie her kennen. Die Position des Betrachters und die gewählte Blickrichtung sind dabei von Bedeutung. Daneben ist noch die Bildweite wichtig, das

heißt die Entfernung des Projektionszentrums vom Rechteck des Bildkoordinatensystems. Bei der Fotografie wäre das die Brennweite. Eine lange Brennweite entspricht einem Teleobjektiv, eine kurze Brennweite dem Weitwinkelobjektiv einer Kamera.

Wie wollen wir nun das Bildkoordinatensystem aufbauen, um die Berechnungen möglichst einfach und schnell zu halten? Unseren Bildschirm könnten wir ja einfach irgendwie und irgendwo in das Realkoordinatensystem setzen. Zur Vereinfachung der Rechenarbeit sollten wir ihn natürlich parallel zu einer Ebene des Objektkoordinatensystems setzen. Das Koordinatensystem unseres Bildschirmes hat nach korrekter Umsetzung seinen Ursprung in der Mitte.

Legen wir nun unseren Bildschirm einfach einmal parallel zur x-z-Ebene in das Realkoordinatensystem. Dann verläuft die Betrachtungsrichtung parallel zur y-Achse des Objektkoordinatensystems.

Eine Transformation der Koordinaten des Objektes in die des Bildes kann durch einfache Verschiebung um den Vektor des Projektionszentrums erfolgen. Den Begriff des Vektors haben Sie ja bereits kennengelernt - ohne ihn werden sie in der Computergrafik kaum noch auskommen. Die perspektivische Transformation, also Umrechnung aller Punkte in die Bildebene, besteht in einem ganz einfachen Rechensatz:

$$x_b = x * \frac{c}{y}$$

$$y_b = z * \frac{c}{y}$$

Dabei stellt c die gewählte Brennweite dar und x_b und y_b sind die errechneten Koordinaten eines Punktes auf dem Bildschirm. Ein »Drahtmodell« unseres Objektes könnte mit dieser Umrechnung bereits auf dem Bildschirm dargestellt werden. Bild 3.10 zeigt die Funktionsweise der Transformation auf das Bildkoordinatensystem.

Die Gleichung einer Ebene im dreidimensionalen Raum lautet in allgemeiner Form:

$$ax + by + cz + d = 0$$

Für das Bildsystem muß dann gelten

$$a'x_b + b'y_b + c'z_b + d' = 0$$

x_b und y_b (die Bildkoordinaten) werden nach den obigen beiden Gleichungen aus x und y ermittelt. Für z_b kann ein Wert der Form

$$z_b = \alpha + \frac{\beta}{z}$$

gefunden werden. α und β lassen sich dabei willkürlich festlegen. Eine wichtige Überlegung spielt dabei die sogenannte Tiefengenauigkeit. Man kann sie durch geeignete Wahl von α und β maximieren. Ein brauchbarer Ansatz ist durch

$$z_b = \frac{-1}{z}$$

zu finden. Man beachte, daß z hier der y-Komponente des Objektkoordinatensystems entspricht.

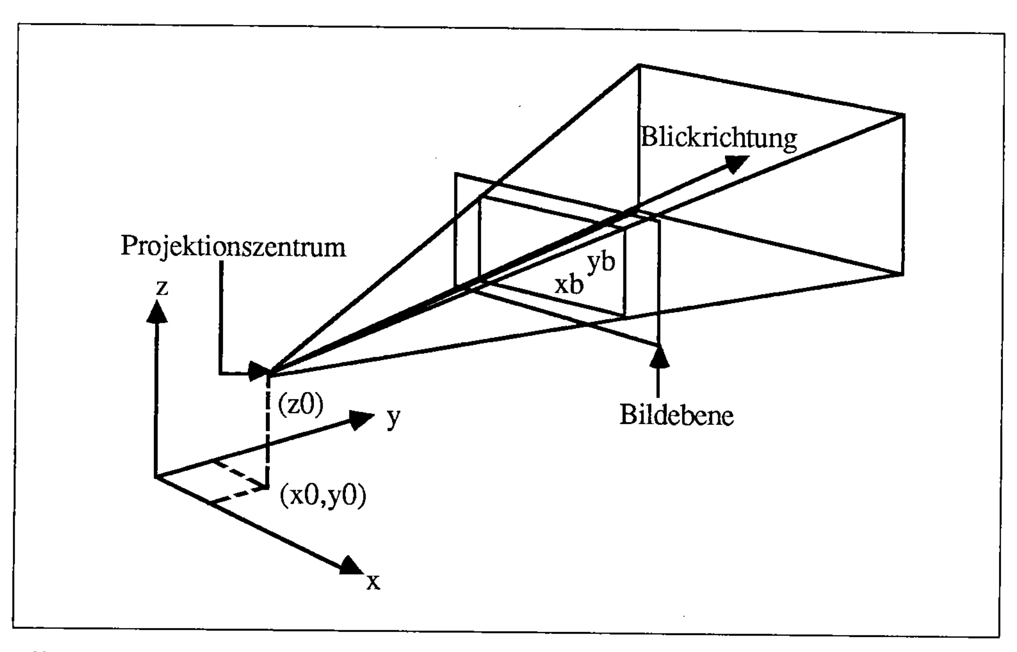


Bild 3.10: Umwandlung vom 3D-Realsystem in Bildkoordinaten

Damit wären wir auch bereits am Ende unsere kleinen Mathematik-Kunde angelangt. Wenn es Ihnen etwas zu kompliziert erscheint, verzweifeln Sie nicht dran, denn damit müssen Sie fertig werden. Computergrafik ist eben nun mal gespickt mit trockener Mathematik.

Buchteil 2

Dieser Buchteil führt Sie zunächst in die Grundlagen der Grafikhardware des Amiga ein. Gedacht ist er für all diejenigen, die wissen wollen, »wie das alles eigentlich funktioniert«. Dabei soll vor allem gezeigt werden, welche grafischen Möglichkeiten die Chips des Amiga bieten und wie die Grafikhardware des Amiga organisiert ist.

Die Grafikhardware wird dann auch in ihren Einzelheiten beschrieben, wobei teilweise einiges Grundwissen vorausgesetzt wird. Für diejenigen, die die Amiga-Chips direkt in Maschinensprache (oder mit POKE-Befehlen) ansprechen wollen, wird auch eine Beschreibung der Register und ihrer Funktionen geliefert.



4 Grundwissen zur Hardware

Die grafische Leistungsfähigkeit des Amiga stammt vor allem von den drei Spezialchips, »Denise«, »Paula« und »Agnus«. Alle drei Chips arbeiten unabhängig vom Hauptprozessor, dem MC68000, was recht ungewöhnlich bei einem preiswerten Computer ist. Wir wollen uns diese Chips und ihr Zusammenwirken nun einmal etwas näher betrachten.

4.1 Die Funktionsweise der Spezialchips

Es gibt viele Computer, die all die anfallenden Arbeiten dem Hauptprozessor überlassen – also die Bearbeitung von Grafikdarstellungen, Musik, Berechnungen und auch die Ein- und Ausgabe. Da all diese Arbeiten der Hauptprozessor übernimmt, müssen sie hintereinander ausgeführt werden. Im Amiga werden diese Tätigkeiten von den drei Custom-Chips ausgeführt; der Prozessor hat nicht mehr viel mit Grafik, Musik und so weiter zu tun. Um diese Verlagerung der Aufgaben noch effektiver zu machen, können die Chips ihre Aufgaben parallel zu denen der CPU (CPU = Central Processing Unit = Hauptprozessor) erledigen. Es handelt sich um sogenannte »Koprozessoren«, die unabhängig vom Hauptprozessor ihrer Arbeit nachgehen.

Die CPU bleibt allerdings immer Herr im Haus; die Betriebssystemsoftware des Amiga wird vom 68000 ausgeführt und steuert die Custom-Chips an. Man kann sich das so vorstellen, daß die CPU eine Aufgabe vorbereitet und den Koprozessor mit allen nötigen Informationen zu seiner Arbeit versorgt. Ist der Koprozessor erst einmal

aktiviert, läuft er von selbst weiter, während sich der Hauptprozessor anderen Dingen zuwenden kann. Deswegen können wir auch Animationsprogramme entwickeln, die den Prozessor fast überhaupt nicht belasten. Einmal gestartet, laufen solche Programme weiter, während der Prozessor schon wieder Berechnungen für unsere nächste 3D-Darstellung bearbeitet.

Ist ein Chip mit seiner Arbeit fertig, signalisiert er das der CPU durch einen sogenannten »Interrupt« und kann von ihr dann erneut mit Arbeit versorgt werden. Alternativ kann die CPU auch »zwischendurch« immer wieder einmal nachschauen, ob ein Koprozessor schon mit seiner Arbeit fertig ist, und darauf entsprechend reagieren. (Dieses Prinzip des »Nachschauens« nennt man auch »Polling«.)

Vielfach werden die Custom-Chips des Amiga auch als DMA-Chips bezeichnet. DMA bedeutet »Direct Memory Access« oder (auf deutsch) »direkter Speicherzugriff«. Die Chips des Amiga sind nämlich direkt an den Speicher angeschlossen und benötigen nicht den Hauptprozessor, um ihnen Daten zuzuführen oder diese nach außen zu geben. Sie arbeiten also wirklich parallel zum Prozessor.

Die DMA-Chips können auf den Speicher über insgesamt 25 sogenannte »DMA-Kanäle« zugreifen. Einige dieser Kanäle sind noch frei für zukünftige Entwicklungen; die meisten sind aber bereits für spezielle Zwecke reserviert. So wird zum Beispiel für jedes Sprite ein DMA-Kanal benötigt – insgesamt also 8 Kanäle. DMA ist ein recht fortschrittliches und teueres Konzept, das erst vor kurzem Eingang in preiswerte Computer gefunden hat. Die Hauptanwendungen der DMA-Technik sind im Amiga Grafik und Klangerzeugung – bei beiden müssen Daten aus dem Speicher teilweise sehr schnell gelesen und in Bilder beziehungsweise Töne umgewandelt werden.

Die DMA-Chips nehmen dem Prozessor aber nicht nur viel Arbeit ab, sondern erledigen diese auch viel schneller, als das der Hauptprozessor könnte. Schließlich sind sie ja speziell – im Gegensatz zum MC68000 – für Grafik und andere Sonderaufgaben entwickelt worden. Trotzdem tritt noch ein kleines Problem auf: Obwohl die Custom Chips unabhängig von der CPU auf den Speicher zugreifen können, tritt manchmal ein Engpaß auf, wenn zwei Chips gleichzeitig mit demselben Teil des Speichers arbeiten wollen. Die RAM-Chips der

heutigen Technologie sind nämlich noch nicht in der Lage, mit mehreren Chips gleichzeitig zu kommunizieren (Solche Chips existieren zwar bereits, sie hätten den Amiga aber wohl um einiges teurer gemacht).

Glücklicherweise kann ein Speicherchip aber mit viel höherer Geschwindigkeit Daten aufnehmen und senden, als der Prozessor das kann. Die Geschwindigkeit, mit der die Chips arbeiten können, nennt man ihre »Taktfrequenz«. Die CPU durchläuft in jeder Sekunde etwa 7,2 Millionen Taktzyklen und hat somit eine Taktfrequenz von 7,2 MHz (Megahertz). In jedem Taktzyklus kann die CPU einen Arbeitsschritt durchführen, also zum Beispiel ein Wort (16 Bit) Daten vom Speicher empfangen oder in ihm ablegen. Der RAM-Speicher läuft nun aber mit einer Geschwindigkeit von 14,4 MHz, durchläuft also in der Sekunde die doppelte Anzahl von Taktzyklen. Diesen Geschwindigkeitsunterschied hat man sich bei der Entwicklung des Amiga zunutze gemacht. Der Prozessor greift nur jeden zweiten RAM-Taktzyklus auf den Speicher zu (wobei er selbst mit voller Geschwindigkeit läuft). Die frei bleibenden Taktzyklen (jeder zweite) werden von den DMA-Chips genutzt. So können also immer abwechselnd der Prozessor und die Koprozessoren auf den Speicher zugreifen.

Nur unter besonderen Umständen benötigen die Spezialchips mehr Taktzyklen als ihnen normalerweise »zugestanden« wird. Besonders zeitintensiv ist zum Beispiel »Blitter«, eine wichtige Komponente innerhalb des Agnus-Chips. Er kann schnelle Datentransfers im Speicher durchführen. In einem solchen Fall (wenn ein Chip mehr Taktzyklen benötigt) wird der Hauptprozessor durch ein bestimmtes Signal kurz angehalten. Die kurze Unterbrechung des Hauptprozessors bringt allerdings keine merkbare Geschwindigkeitsverringerung mit sich. Die dadurch entstehende Geschwindigkeitssenkung des Prozessors ist nämlich Mittel zum Zweck, die Gesamtgeschwindigkeit zu erhöhen. Schließlich sind die Spezialchips bei den Aufgaben, für die sie gebaut sind, schneller als der Hauptprozessor selbst. Die langsamere Einheit muß sich also immer der schnelleren unterordnen.

4.2 Die Chips – ein kurzer Überblick

Aber wozu erzähle ich Ihnen das alles, wenn Sie doch nur Grafik programmieren wollen? Ganz einfach: Zur Programmierung eines Computersystems muß man auch seine internen Zusammenhänge kennen. Einen Fernseher kann man auch erst dann bedienen, wenn man weiß, welcher Knopf welchen Zweck hat. Wer versucht, die Helligkeit am Lautstärkeregler zu ändern, ist selbst schuld, daß er sich nicht ausreichend informiert hat.

Die Namen der drei Chips leiten sich auf nicht ganz nachvollziehbare Weise von ihren Hauptanwendungsgebieten ab. Der Name »Paula« stammt beispielsweise von »Peripheral/Audio«. Ebenso leitet sich der Name des Chips »Agnus« von »Adress Generator« ab. Und schließlich gibt es da noch »Denise«, was sich von »Display Encoder« ableitet. Alle drei Chips zusammen bilden eigentlich eine große Einheit. Mit der heute zur Verfügung stehenden Technologie konnte man sie jedoch nicht auf einem Chip zusammenfassen. Man muß sie sich jedoch als einen einzigen Chip vorstellen, denn so eng arbeiten sie zusammen. So ist zum Beispiel der Löwenanteil der sogenannten Playfield-Hardware (siehe dazu auch die folgenden Kapitel) im Denise-Chip enthalten, ein anderer Teil aber auch in Agnus.

4.2.1 Paula

Der erste der genannten Chips, Paula, ist für die Grafikprogrammierung relativ uninteressant. Paula ist zuständig für die Klangerzeugung, die Verwaltung der Ein- und Ausgabe, sowie Interruptverarbeitung (siehe dazu auch Markus Breuer, *Das Amiga-Handbuch*, Seite 364).

4.2.2 Agnus

Der Agnus-Chip ist für unsere Interessen in grafischer Hinsicht weit mehr von Bedeutung. Er beinhaltet unter anderem den Adreß-Generator, der den Zugriff aller Koprozessoren über die DMA-Kanäle auf den RAM-Speicher steuert. Zwei Koprozessoren inner-

halb des Agnus-Chips, der Blitter und der Copper, tragen besonders zu schnellen bewegten Grafiken bei.

Um diese schnelle Grafikverarbeitung überhaupt erst zu ermöglichen, mußte nämlich eine Einheit geschaffen werden, die schnelle Manipulationen von Daten im Frame-Buffer (die der Videochip in ein Bild umwandelt) im RAM-Speicher möglich macht. Diese Einheit ist der Blitter. Sein Name beruht auf einer Prozedur zum Verschieben und Kombinieren von Speicherblöcken für die Benutzung dieser Speicherblöcke als Grafik. Diese Prozedur war als Befehl namens »BitBlt« in einem älteren Computer der Firma Xerox installiert. Da die für den Amiga entwickelte Hardwareeinheit genau dasselbe wie diese Prozedur (und einiges darüber hinaus) leistet, lag es nahe, ihr auch einen ähnlichen Namen zu geben: eben »Blitter«. Mit Hilfe des Blitters ist es zum Beispiel möglich, Flächen mit einer Geschwindigkeit von 1 Million Bildpunkten pro Sekunde zu füllen oder mit derselben Geschwindigkeit gerade Linien zu ziehen. Die Hauptanwendung des Blitters liegt aber bei der schnellen Grafik-Animation. Dabei bewegen sich einzelne Objekte (Teile eines Bildes) über den Bildschirm, als wären es einzelne Bilder. Diese Objekte sind aber natürlich nichts anderes als Datenblöcke im Frame-Buffer, die durch den Blitter sehr schnell im Speicher verschoben werden können, was dann zu dem Eindruck einer Bewegung führt.

Statt Daten einfach nur zu bewegen, kann der Blitter sie aber auch verknüpfen. Das schlägt sich dann zum Beispiel in der Tatsache nieder, daß ein Objekt zum Teil transparent sein kann. Der Blitter verknüpft dazu die Hintergrundgrafik mit dem Bild des bewegten Objekts. Die so bewegbaren Objekte können größer sein als Sprites. Es handelt sich um sogenannte »BOBs«. BOB ist eine Verballhornung von »Blitter Object« und verhält sich wie ein durch Software erzeugtes Sprite.

Ein weiterer Coprozessor innerhalb des Agnus-Chips ist der Copper. Copper steht für »Co-Prozessor«; einen deskriptiveren Namen hat man wohl nicht gefunden. Der Copper ist allerdings ein ganz besonderer Coprozessor. Er ist eng an den Elektronenstrahl gekoppelt, mit dem das Bild auf die Phosphorschicht des Monitors gezeichnet wird. Der Copper hat die Fähigkeit, zu erkennen, in welcher Zeile und Spalte des Bildes (also bei welchen Koordinaten) dieser Strahl gerade ist. Er kann zum Beispiel auf das Erreichen einer bestimmten

Position warten, andere Chips beeinflussen oder spezielle Programme starten. Durch diese Fähigkeit ist zum Beispiel auch das Konzept der virtuellen Bildschirme möglich, die man einfach mit der Maus nach oben oder unten ziehen kann. Hat der Rasterstrahl die Stelle erreicht, an der ein neuer Bildschirm beginnt, werden die entsprechenden Register in Grafikchip Denise geändert. Am unteren Rand des Monitorbildes hält der Rasterstrahl an und beginnt dann wieder links oben, worauf der Copper erneut die Register ändert. Das Betriebssystem des Amiga steuert den Copper in sogenannten »Copperlisten«, die man selbst programmieren kann. Durch geschickte Programmierung in Maschinensprache kann man phantastische Grafikeffekte aus dem Copper herausholen.

4.2.3 Denise

Denise ist der für die Grafikprogrammierung wohl wichtigste Chip des Amiga . Denise ist verantwortlich für die Umsetzung von Daten des RAM-Speichers (Frame-Buffers) in das Bild, das hinterher auf dem Monitor sichtbar ist. Denise kann die Daten dabei auf verschiedene Arten bearbeiten. Diese Arten nennt man auch »Grafikmodi« (Singular »Grafikmodus«). Jeder Grafikmodus hat eine bestimmte Auflösung und eine bestimmte Anzahl von möglichen Farben. Denise verfügt über Register (falls Sie es immer noch nicht wissen sollten: Register sind Speicherstellen, deren Inhalt den Chips sagt, wie sie sich zu verhalten haben), die die horizontale und vertikale Auflösung bestimmen, die Anzahl der Bitplanes, die die 32 möglichen Farben festlegen.

Neben Farben und Grafikauflösungen bietet Denise aber noch weitere Möglichkeiten. Die Sprite-Logik und die Playfield-Hardware bringen Bewegung ins Bild. Wie Sprites (hardwaregesteuerte bewegbare Objekte) und Playfields (das sind Grafiken, deren Größe unabhängig von der Bildschirmgröße und Grafikauflösung sind) genau funktionieren, werden Sie in den folgenden Kapiteln noch sehen.

5

Bitmapgrafiken – grundlegende Organisation einer Grafik

Die Grafik des Amiga ist »bitmapped«. Wörtlich übersetzt ist eine Bitmap eine »Speicherlandkarte«. So ungefähr wie eine Landkarte kann man sich eine Bitmap auch vorstellen: Ein Punkt, der auf dem Bildschirm gesetzt sein soll, entspricht einem gesetzten Bit (also einer 1) in der Bitmap. Ein Punkt der nicht gesetzt sein soll, enthält eine 0. Auf dem Bildschirm wird das erreicht, indem der Rasterstrahl, der zickzackförmig von links oben nach rechts unten wandert, seinen Elektronenstrahl verstärkt und somit einen leuchtenden Punkt erzeugt, wenn der Videochip (Denise) im Speicher auf eine 1 stößt. Bei einer Null im Speicher erhält der Strahl, was einen dunklen Punkt ergibt. Der Videochip geht dabei Bit für Bit im Speicher vor. Jeder Bildpunkt entspricht also grundsätzlich einem Bit im Speicher.

Der Speicher eines Computers beginnt von Null an aufwärtszählend, ist also geradlinig. Der Bildschirm ist aber zeilenweise aufgebaut. Bei einer Auflösung von 320 mal 200 sieht dann die Speicherorganisation ungefähr so aus wie in den Bildern 5.1 und 5.2:

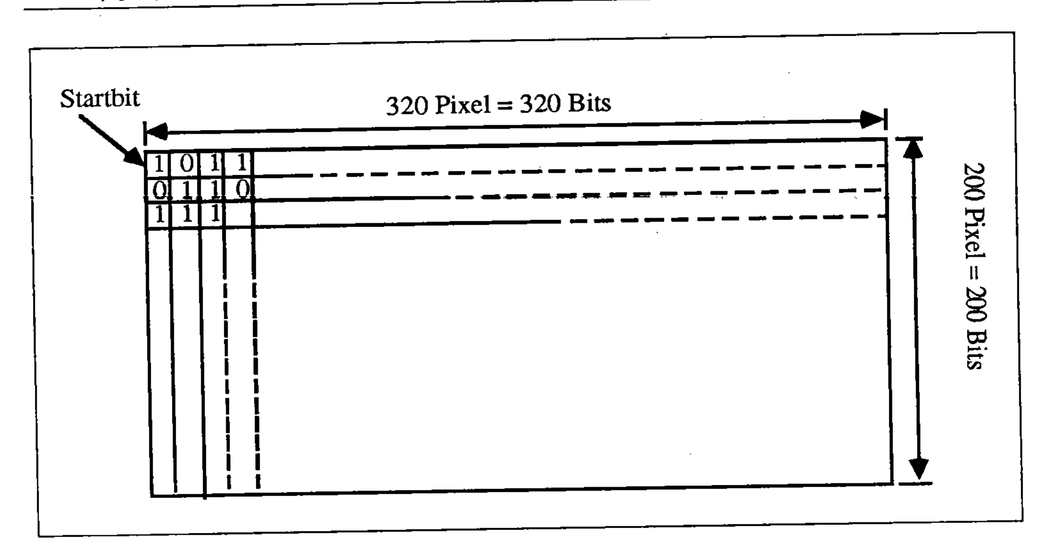


Bild 5.1: Eine Bitmap

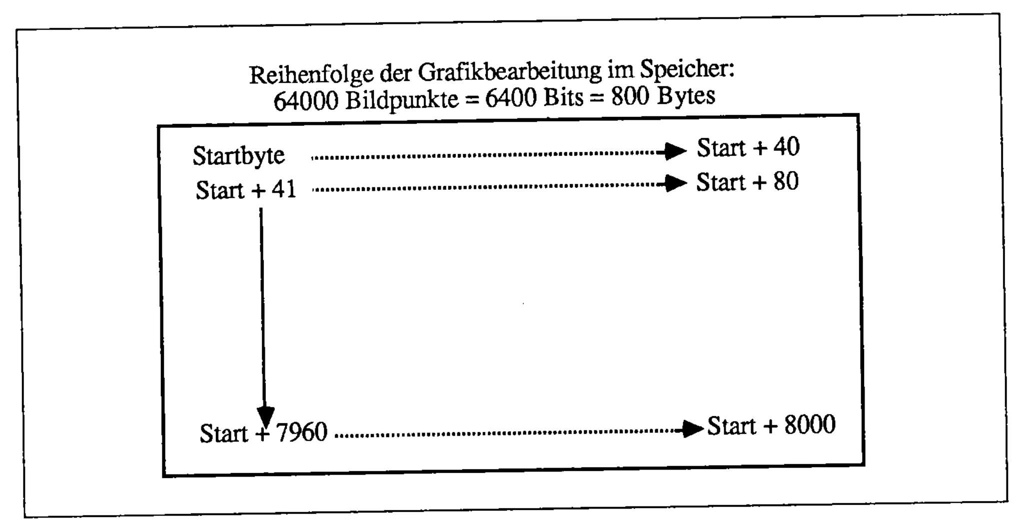


Bild 5.2: Die Organisation einer Grafik im Speicher

»Schön und gut«, werden Sie jetzt vielleicht sagen, »aber wo bleibt die Farbe?« Es gibt mehrere Möglichkeiten, Farben in einer Bitmap zu kodieren. Wie bei Computern üblich, wird die Farbkodierung wieder mit Hilfe von Bits erledigt. Bits können zwar nur gleich Null oder Eins sein; das muß uns aber nicht unbedingt auf zwei Farben beschränken. Es werden einfach mehrere Bits zusammengenommen,

um eine bestimmte Farbe darzustellen. Je mehr Bits wir zur Angabe einer Farbe benutzen, desto mehr Farben können wir verwenden. Dabei gilt: Je nach Anzahl *n* der verwendeten Bits sind 2^n Farben möglich.

Wie organisieren wir jetzt diese Bits ordentlich im Speicher? Wir könnten sie direkt hintereinandersetzen, also immer ein Bitpaar im Speicher ergibt einen Punkt. So ist Farbgrafik beispielsweise im C64 realisiert. Allerdings sind wir mit dieser Technik relativ beschränkt. Die Auflösung der Grafik des C64 halbiert sich, wenn wir auf den Mehrfarbmodus umschalten. Würden wir mit dieser Technik noch mehr Farben darstellen wollen, würde die Auflösung auch immer kleiner werden. Natürlich gibt es noch einige andere Techniken; sobald aber die Anzahl der möglichen Farben frei bestimmbar sein soll (bis zu einem gewissen Limit, versteht sich), ist die Methode, die im Amiga verwendet wird, noch die sinnvollste. Die Amiga-Entwickler wollten einerseits möglichst viele Farben darstellen, andererseits aber keine Grafikauflösung verlieren. Zusätzlich sollte der Speicherbedarf geringer sein, wenn sich der Programmierer oder Benutzer entschließt, weniger Farben zu benutzen.

Die Speicherorganisation von farbiger Grafik im Amiga ist so ausgelegt, daß nicht mehrere im Speicher hintereinanderliegende Bits für einen Punkt verwendet werden, sondern »übereinander« liegende. »Übereinanderliegend« soll in diesem Fall heißen, daß man mehrere Bitmaps gleicher Größe übereinanderlegt. Die Kombination der übereinander liegenden Bits gibt dann eine Zahl beziehungsweise einen Farbwert für einen Punkt auf dem Bildschirm.

Die Zahl, die sich aus der Kombination der Bildpunkte ergibt, ist zugleich die Nummer des entsprechenden Farbregisters. Der Amiga hat 32 solche Farbregister. In einem »normalen« Grafikmodus können wir also höchstens 32 Farben gleichzeitig verwenden.

Legen wir zwei Bitplanes übereinander, können wir (wir erinnern uns: n Bits = 2^n Farben) vier Farben verwenden. Legen wir drei Bitplanes übereinander, haben wir acht Farben. Im allerhöchsten Fall können wir fünf Bitplanes miteinander verknüpfen und können dann zwischen 32 Farben wählen.

Die Farbregister beinhalten Informationen über die jeweilige Farbe. Jedes Farbregister ist 12 Bits breit. Jede Farbe besteht aus gewissen

Anteilen rot, grün und blau. Innerhalb dieser 12 Bits sind jeweils vier Bits für die Rot-Intensität, vier Bits für Grün und vier Bits für Blau vorgesehen. Jede Farbkomponente kann also in 16 Abstufungen der resultierenden Farbe beigemischt werden. In jedem Farbregister kann somit eine Kombination von 16 mal 16 mal 16 Farbabstufungen dargestellt werden – insgesamt also 4096 Farben.

Wie das Bitmap-Prinzip bei Verwendung von drei Bitmaps aussieht, sehen Sie in Bild 5.3.

Diese normalen Bitmap-Grafiken sind aber nur eine Teilmenge der Grafikfähigkeiten, die uns die Playfield-Hardware beschert, die wiederum zum Großteil Bestandteil des Denise-Chips ist. Wie Sie bereits gesehen haben, spielen die Register der Hardware eine große Rolle bei der Darstellung von Grafiken. Nicht nur die Farben, sondern auch Speicheradressen, wo unsere Grafik liegt, Grafikmodi (Auflösung) und so weiter werden in Registern festgelegt. Jedes dieser Register hat einen Namen und eine fixe Speicheradresse.

In den folgenden Kapiteln wird jeweils nur der Name des Registers genannt. Die Register werden am Ende eines jeden Kapitels kurz zusammengefaßt. Im Anhang finden Sie dann auch noch eine Zusammenfassung aller für Grafik notwendigen Register (inklusive ihrer Speicheradressen).

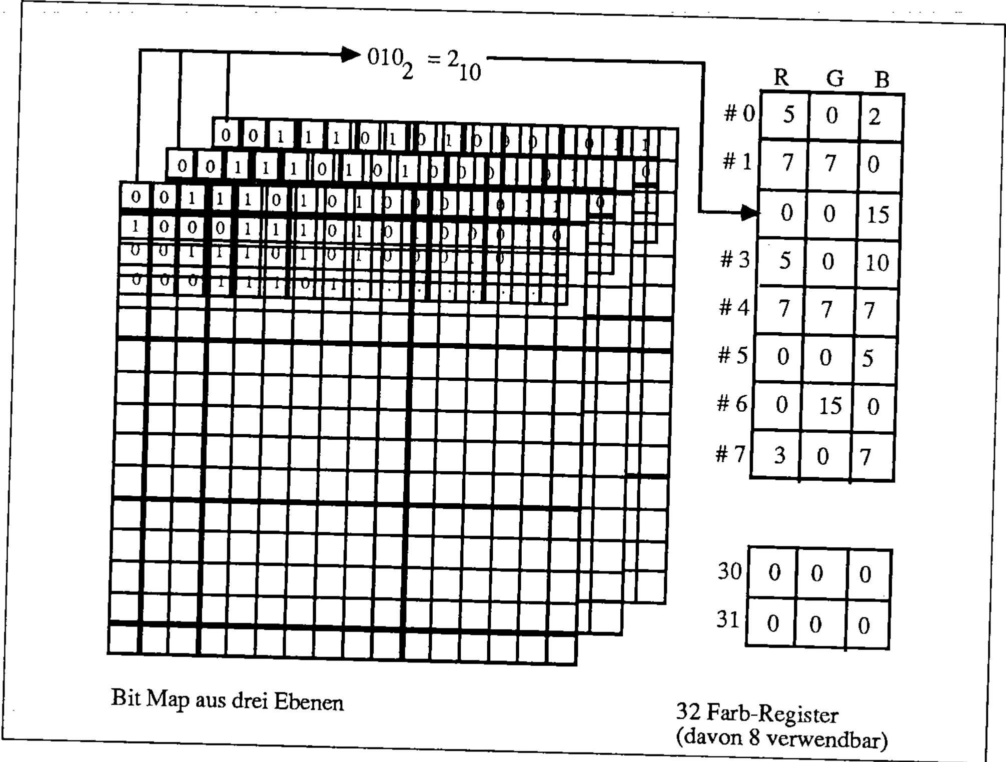


Bild 5.3: Die Zuordnung von Bitplanes zu Farben

6 Die Playfield-Hardware

Übereinandergelegte Bitplanes, die der Größe des Bildschirms (also seiner Auflösung) entsprechen, nennt man auch »BASIC Playfields«. Ein Playfield ist also eine Ansammlung von einer oder mehreren Bitplanes, die zusammen ein Grafikbild ergeben. Mit diesen Playfields (auf deutsch »Spielfeldern«) werden wir uns in diesem Kapitel beschäftigen.

6.1 Einfache Playfields

Um solche »einfachen« Playfields (»BASIC Playfields«) zu erzeugen, muß man zuerst entscheiden, welche Auflösung man wünscht, welche und wieviele Farben benötigt werden, wieviele Bitplanes man dafür verwenden will und in welchen Speicherbereich der Frame-Buffer liegt.

6.1.1 Bitplanes und Farben

Zur Entscheidung über die Zahl der nötigen Bitplanes eine kurze Übersichtstabelle.

Gewünschte Anzahl Farben	Nötige Anzahl Bitplanes
1-2	1
3-4	2
5-8	3
9-16	4
17-32	5

Die Farbregister werden zunächst, wie bereits beschrieben, mit Werten für die RGB-Anteile gefüllt. Sie besitzen die Namen COLOR0, COLOR1 und so weiter bis COLOR31. Dabei bestimmt COLOR0 immer die Hintergrundfarbe. Die Hintergrundfarbe ist die Farbe, die sich an jeder Stelle des Bildschirms befindet, an der kein anderes Objekt vorhanden ist, und wird auch außerhalb des definierten Display-Bereiches (am Rand) angezeigt.

Wird ein Genlock-Interface angeschlossen, ersetzt das durch Kamera, Videorekorder oder Laserdisk produzierte Bild allerdings die in COLOR0 angegebene Hintergrundfarbe.

Von den 16 Bits, die in jedem Farbregister zur Verfügung stehen, werden allerdings nur 12 (4 für jeden Farbanteil) genutzt: die Bits 15-12 bleiben unbenutzt, die Bits 11-8 bestimmen den Rotanteil, die Bits 7-4 den Grünanteil und die Bits 3-0 schließlich den Blauanteil.

Die Farbregister können nicht gelesen werden, es handelt sich um sogenannte »Write Only«-Register. Aus diesem Grunde sollte man, will man Farbmanipulationen vornehmen, bei denen die Kenntnis der bisherigen Werte notwendig ist, eine Kopie der Inhalte an anderer Stelle ablegen. Sobald man in ein Farbregister schreibt, sollte man denselben Wert auch in die RAM-Sicherheitskopie schreiben. So weiß man immer, welcher Wert sich in welchem Farbregister befindet.

Sind die Farben gesetzt, muß dem Computer als Nächstes die Anzahl der benötigten Bitplanes mitgeteilt werden. Der Denise-Chip benutzt ein multifunktionales Register BPLCON0 (*Bit Plane Control Register 0*). »Multifunktional« bedeutet, daß das Register nicht nur auf eine Funktion (wie zum Beispiel Farbe) beschränkt ist. In mehrfunktionalen Registern sind einzelne Bits für bestimmte Zwecke

vorgesehen. In unserem Fall bestimmen die Bits 14, 13 und 12 die Anzahl der verwendeten Bitplanes. In multifunktionalen Registern erhält jedes Bit ebenfalls einen Namen. Unsere drei Bits wurden von den Amiga-Entwicklern BPU2, BPU1 und BPU0 genannt, wobei BPU für *Bit Planes Used* (benutzte Bitplanes) steht. Zusammen bilden die drei Bits eine dreistellige Binärzahl, die die Anzahl der Bitebenen angibt. Die Bitkombination 001 entspricht 1 Bitebene, 010 (dezimal 2) entspricht zwei Bitebenen, 011 drei Bitebenen und so weiter bis zur Bitkombination 110, die sechs Bitebenen aktiviert. (Die Bitkombinationen 000 und 111 werden nicht benutzt.)

Die Kombination 110 ergibt die Zahl 6. Wir haben allerdings bereits gesagt, daß die Videologik nur 5 Bitplanes verarbeiten kann. Die sechste Bitplane wird nicht direkt angezeigt, sondern in einigen Grafikmodi als Hilfs-Bitplane verwendet. Die Grafikmodi, die eine sechste Bitplane benutzen, sind der Halfbrite-Modus, der Hold & Modify-Modus und der Dual-Playfield-Modus. Um diese Grafikmodi zu aktivieren, müssen Sie sechs Bitplanes angeben und anschließend noch dem Grafikmodi entsprechend weitere Register ändern.

Die Kombination 111 wird nicht benutzt; sie ist für zukünftige Erweiterungen gedacht. Sie werden auch in späteren Kapiteln bemerken, daß viele Bits oder Bitkombinationen in den Registern nicht benötigt werden. Damit haben sich die Entwickler des Amiga ein Hintertürchen geschaffen, um ohne großen Aufwand technische Verbesserungen und Erneuerungen zu schaffen, ohne mit den neuen Chips inkompatibel zur alten Software zu werden. Wird zum Beispiel eine neue Version der Playfield-Hardware entwickelt, die zur Darstellung eines neuen Grafikmodus sieben Bitplanes benötigt, kann die hier freie Bitkombination benutzt werden; neue Register müssen nicht eingebaut werden, und so bleibt die Software kompatibel.

In multifunktionalen Registern wie BPLCON0 können keine einzelnen Bits gesetzt werden. Wir müssen also das gesamte Speicherwort neu setzen. Aus diesem Grunde ist es wichtig, die Inhalte der zu anderen Funktionen gehörenden Bits zu wissen. Auch hier gilt wieder: immer eine Sicherheitskopie im RAM-Speicher ablegen.

6.1.2 Grafikauflösungen und einfache Grafikmodi

Bevor wir ein Bild produzieren, müssen wir zuerst seine Auflösung festlegen. Zur Verfügung stehen uns folgende Grafikauflösungen: 320 oder 640 Punkte horizontal und 200 oder 400 Punkte vertikal. Die zwei horizontalen Auflösungsmöglichkeiten nennen wir »LoRes« und »HiRes«. Zur Festlegung der horizontalen Auflösung verwenden wir eine weitere Funktion des Registers BPLCON0. Dazu genügt uns ein Bit, da wir ja nur zwei Zustände unterscheiden müssen. Bit 15 des BPLCON0-Registers heißt HIRES, da wir damit quasi die hohe Auflösung (HiRes = High Resolution) an- beziehungsweise abschalten können. Ist das Bit gesetzt, hat das Playfield eine horizontale Auflösung von 640 Punkten; ist das Bit nicht gesetzt, beträgt die Auflösung 320 Punkte.

Im höherauflösenden Modus hat die Videologik allerdings auch doppelt so viel Speicher zu verwalten. Da mit der derzeitigen Speichertechnologie jedoch die Daten nicht schnell genug aus dem Speicher geholt werden können, stehen im hochauflösenden Modus nur höchstens vier Bitplanes und damit 16 Farben zur Verfügung. (Außer, wir tricksen den Computer aus – aber das ist ein Kapitel für sich).

Die vertikale Auflösung beträgt im Normalfall 200 Zeilen, beim PAL-Amiga mit der Betriebssystemversion 1.2 256 Zeilen. Durch den sogenannten »Interlace«-Modus können wir die Auflösung auf 400 beziehungsweise 512 Zeilen verdoppeln.

Was bedeutet nun »Interlace«? 200 beziehungsweise 256 Punkte ist die Grenze, die von einigermaßen erschwinglichen Monitoren und Farbfernsehern gesetzt wird, ohne das Bildflimmern auftritt. Die Bildwiederholfrequenz beträgt bei dieser Auflösung 60 Hz. Das heißt, 60 mal pro Sekunde wird das Bild neu gezeichnet – schneller als das Auge einzelne Bilder trennen könnte. Das englische Wort »Interlace« sagt uns schon fast, wie wir unsere Auflösung erhöhen: Es bedeutet »verschränkt« oder »überlagert«.

Das funktioniert folgendermaßen: Der Bildschirm wird in doppelt so viele Zeilen aufgeteilt, indem zwei Playfields gezeigt werden. Das zweite Playfield wird um eine (neue, schmalere) Zeile nach unten verschoben. Durch die Verschiebung liegen nun zwei Punkte übereinander, wo vorher nur einer lag. Da die Videologik jetzt zwei

Playfields verarbeiten muß, arbeitet sie nur noch mit halber Geschwindigkeit. Das Bild wird nun mit einer Bildwiederholfrequenz von effektiv 30 Hz angezeigt, was das Flimmern (leider) deutlich wahrnehmbar macht. Dieser Nachteil wird jedoch dadurch aufgewogen, daß Linien oder andere Objekte bei der höheren Auflösung nicht mehr am »Treppen«-Effekt leiden.

Die Funktionsweise des Interlace-Modus erkennen Sie in Bild 6.1. Es wird jeweils die erste Zeile des ersten Playfields dargestellt, dann die erste Zeile des zweiten Playfields, die zweite Zeile des ersten Playfields und so weiter.

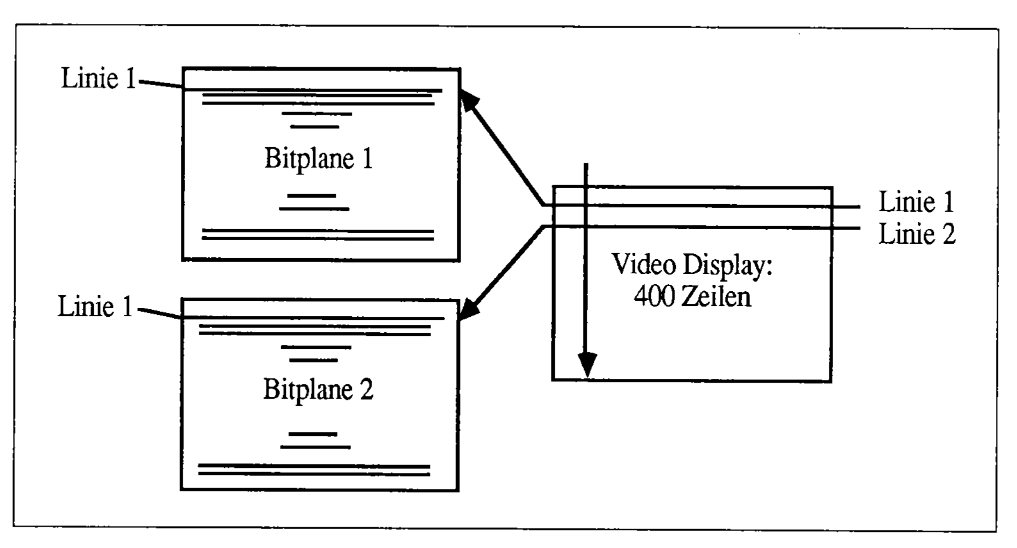


Bild 6.1: So funktioniert der Interlace

Benutzen wir die Fähigkeit des Blitters, Linien zu ziehen oder Polygone zu malen, im Interlace-Modus, erkennt dieser selbsttätig, daß der Interlace-Modus angeschaltet ist. Der Blitter behandelt unsere Interlace-Grafik also nicht als zwei separate Felder, sondern als ein einziges Display.

Zur Festlegung des Interlace-Modus verwenden wir eine weitere Funktion des Registers BPLCON0. Bit 2, auch LACE genannt, bestimmt, welche vertikale Auflösung das Bild hat. Wie auch schon bei der Unterscheidung HiRes/LoRes bedeutet auch hier ein gesetztes Bit (1) ein Anschalten des Interlace. Ein nicht gesetztes

LACE-Bit ergibt die normale Auflösung von nur 200 beziehungsweise 256 Punkten.

Der Interlace-Modus verlangt von uns allerdings etwas mehr Programmierarbeit als die geringere Auflösung, da hier einige Register mehr gesetzt werden müssen (schließlich haben wir es mit mehr als nur einem Playfield zu tun; für jedes Playfield muß der Speicher reserviert werden, in dem der entsprechende Frame-Buffer liegt).

6.1.3 Festlegen eines Speichers für die Grafik

Und da wären wir bereits beim Hauptproblem: dem Speicher. Die Hardware des Computers soll ja nicht einfach zeigen, was sich in irgendwelchen Speicherstellen befindet. Deshalb benötigen die Chips Angaben, in welchem Speicherbereich sich der Frame-Buffer (die Bitplanes) für unsere Grafik befindet. Das Playfield besteht dabei aus mehreren übereinandergelagerten Bitplanes. Um der Videologik den richtigen Speicher zu zeigen, setzen wir wieder eine Reihe von Registern. Die »Startadresse« ist ein 16-Bit-Wort, das die Speicheradresse angibt, die der linken oberen Ecke des Bildes entspricht.

Je nach Grafikauflösung benötigen wir entsprechend mehr oder weniger Speicher. Manchmal muß man sich gegen eine höhere Auflösung zugunsten des geringeren Speicherplatzbedarfs entscheiden, da das Hauptprogramm auch noch in den Speicher passen muß. Die folgende Tabelle zeigt, welche Auflösung wieviel Speicher benötigt:

Auflösung	Grafikmodus	Bytes /Bitplane
320 x 200	LoRes, Non-Interlace	8000
320 x 400	LoRes, Interlace	16000
640 x 200	HiRes, Non-Interlace	16000
640 x 400	HiRes, Interlace	32000

Tabelle 6.1: Grafikmodi und der von ihnen benötigte Speicherplatz

Ein Beispiel: Wir wollen ein 8farbiges Bild mit einer Auflösung von 640 mal 200 Punkten auf den Bildschirm bringen. Das Bild besteht, wie wir wissen, aus drei Bitplanes. Jede dieser Bitplanes benötigt

16 000 Bytes. Der Speicher muß für jede Bitplane fortlaufend sein, darf also keine Unterbrechungen haben. Für ein solches Bild brauchen wir drei in sich abgeschlossene Speicherblöcke von jeweils 16 KByte, insgesamt also 48 KByte. (Wie eine Grafik im Speicher organisiert ist, wurde bereits im Einführungskapitel beschrieben.)

Die Angabe des reservierten Grafikspeichers nehmen wir durch das Setzen von jeweils zwei Adreßregister pro Bitplane vor. Insgesamt gibt es 12 Adreßregister. Die Register heißen BPLxPTH und BPLxPTL. Ganz ausgesprochen heißen diese Abkürzungen »Bitplane x Pointer High« und »Bitplane x Pointer Low«. Das x ist die Nummer der Bitplanes (von 1 bis 6). Wollen wir beispielsweise Bitplane 1 einen Speicher zuweisen, so schreiben wir seine Adresse in das Registerpaar BPL1PTH und BPL1PTL. Registerpaare, die mit Endungen PTH und PTL benannt werden, geben immer 19-Bit-Adressen an (19 Bits deshalb, weil die Custom-Chips nur auf die unteren 512 KByte zugreifen können – dazu genügen 19 Bits). Maschinensprache-programmierer können diese Register aber auch als 32-Bit-Adresse behandeln und sie mit Longwords (ein Longword = 32 Bit) füllen. Dabei schreibt man das Longword jeweils in die Adresse des **obersten** Registerwortes (dessen Name mit PTH endet).

Beachten Sie bitte, daß diese Speicherzuweisungen nur für die Playfields gelten. Andere Grafikelemente benötigen ebenfalls Speicherplatz. Speicherzuweisungen für Sprites und BOBs werden in den dazugehörigen Kapiteln behandelt.

6.1.4 Das Display-Window

Im Gegensatz zu herkömmlichen Computern können wir beim Amiga nicht nur die Grafikauflösung bestimmen, sondern auch, welcher Ausschnitt der Grafik gezeigt wird. Wir müssen nicht unbedingt immer die gesamte Grafik zeigen; der Teil, den wir auf dem Bildschirm darstellen, nennt sich auch »Display-Window« und kann ein kleiner Ausschnitt aus einem viel größeren Bild sein. Alles, was sich außerhalb des Display-Windows befindet, wird nicht angezeigt.

Wie so ein Display-Window auf dem Bildschirm aussieht, zeigt Bild 6.2. Im Normalfall hat das Display-Window jedoch dieselbe Größe wie die Grafikauflösung. Im Falle der Grafik aus Abbildung 6.2 haben wir aber ein Playfield genommen, das größer als die nor-

male Bildschirmgröße ist (dazu später mehr). Das Display-Window erlaubt uns nun zum Beispiel, in einer riesigen Grafik beliebige Ausschnitte zu betrachten. Das erzeugt den sogenannten »Lupeneffekt«: Man denkt, man sieht mit einer Lupe auf ein riesiges (Grafik)-Gebiet.

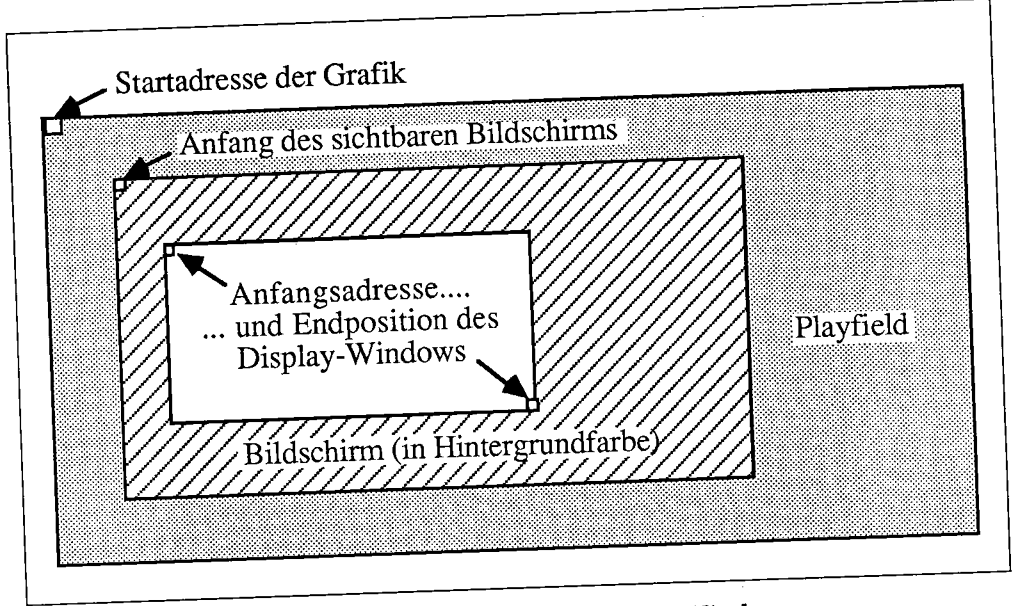


Bild 6.2: Der Lupen-Effekt durch ein kleines Display-Window

Die Größe des Display-Windows wird festgelegt, indem wir die horizontalen Start- und Endkoordinaten angeben. Diese Koordinaten beziehen sich jedoch nicht unbedingt auf die Grafikauflösung! Wir berechnen die Koordinaten des Display-Windows wie folgt: Die vertikale Auflösung ist für die Display-Window-Register immer eine Rasterzeile (bei Interlace werden die Halbbilder um je eine **halbe** Rasterzeile verschoben angezeigt!). Die horizontale Auflösung, die die Display-Window-Register anerkennen, bezieht sich zudem auf niedrigauflösende Pixels. Haben wir Hires-Modus angeschaltet, müssen wir also, wenn wir die horizontale Start- und Endposition des Display-Windows angeben wollen, immer die Bildschirmkoordinaten des jeweiligen Punktes durch zwei teilen.

Obwohl die Koordinaten der linken oberen Ecke des Bildschirms bei (0,0) beginnen, ist die erste vertikale Position des Display-Windows

normalerweise auf 44 ($2C) und die erste horizontale Position auf 129 ($81) eingestellt. Diese Einstellung von ($81,$2C) beruht auf der Tatsache, daß viele Videomonitore so gebaut sind, daß am Rand des Bildschirms ein unscharfes Bild entstehen kann. Um dies zu vermeiden, bestreicht die Videologik einen Bereich mit dem Elektronenstrahl, der größer ist als die üblicherweise maximal verwendeten 320x200 LoRes-Punkte (dies nennt man »Overscan«). Die meisten Programme benutzen davon aber nur ein Rechteck, das deutlich kleiner als der ganze Bildschirm ist, für ihre Grafikausgaben. Ein Programm kann auf Wunsch aber auch den ganzen Bildschirm nutzen. (DELUXE PAINT 2 nutzt zum Beispiel diese Möglichkeit.) Grafik 6.3 zeigt den Zusammenhang zwischen Display-Window, sichtbarem Bildschirm und dem Bereich, der vom Elektronenstrahl benutzt wird.

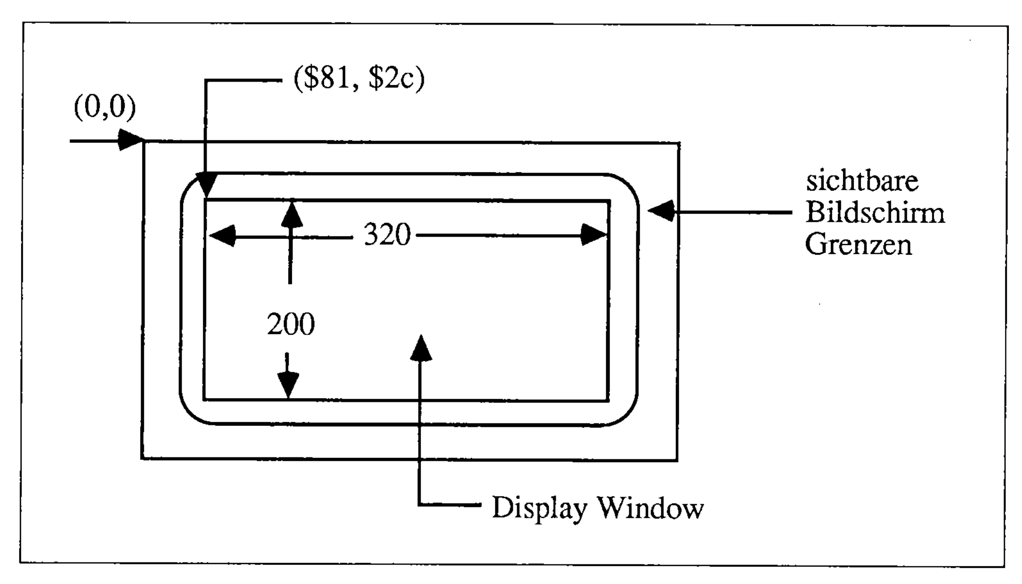


Bild 6.3: Der Overscan-Effekt

Das Register, das dem Videochip Auskunft über die linke obere Ecke des Display-Windows gibt, heißt DIWSTART (*Display Window Start*). Dieses eine Register beinhaltet sowohl die x- als auch die y-Koordinate. Die Koordinaten sind, wie bereits betont, in einem Koordinatensystem angegeben, das dem LowRes-Non-Interlace-Modus entspricht. Da das Register 16 Bits breit ist, benutzt es jeweils ein

Byte (8 Bits) für die x- und ein Byte für die y-Koordinate. Diese Bytes nennen wir HSTART und VSTART. Da 8 Bits höchstens einen Wert von 256 ergeben, können wir die x-Koordinate der Startposition nicht weiter als in das vierte Fünftel des Bildschirms schieben. Diese Beschränkung kann uns allerdings relativ egal sein, weil es sowieso unsinnig ist, nur das rechte Fünftel des Bildschirms zu benutzen. Da die y-Koordinate normalerweise sowieso nur bis 200 geht, wird uns durch die 8-Bit-Notation hier überhaupt keine Beschränkung auferlegt.

Haben wir die Startposition gesetzt, muß noch die rechte untere Ecke des Display-Windows festgelegt werden. Diese wird durch das Register DIWSTOP (*Display Window Stop*) kontrolliert. Auch hier ist die Koordinatenangabe durch zwei Bytes, HSTOP und VSTOP, realisiert. Da wir allerdings höchstens einen Wert von 256 auf diese Weise darstellen können, haben die Entwickler des Amiga die Videologik so gebaut, daß zum Wert HSTOP automatisch die Zahl 256 addiert wird. Wollen wir eine Koordinate in dieses Register schreiben, müssen wir also immer erst 256 vom gewünschten x-Wert abziehen und den so errechneten Wert in das Register schreiben. Der normale HSTOP-Wert ist $1C1, wird dann also als $C1 in das Register geschrieben.

Die VSTOP-Position ist auf die untere Hälfte des Bildschirms beschränkt. Dies wird bewirkt, indem die Hardware das höchstwertige Bit der Stop-Position zum Komplement des nächsten höchstwertigen Bits macht. Das erlaubt VSTOP-Positionen, die größer als 256 sind. Der VSTOP-Normalwert liegt bei $F4.

Die Register DIWSTART und DIWSTOP müssen nicht unbedingt vom Programmierer gesetzt werden. Dies ist nur nötig, wenn man ein Display-Window wünscht, das von dem normalerweise verwendeten (zum Beispiel dem in der Workbench) abweicht. Normaler Inhalt für DIWSTART ist $2C81 und für DIWSTOP $F4C1.

6.1.5 Wie die Daten angezeigt werden

Das Anzeigen eines Bildes ist für die Hardware kein Kinderspiel. Es müssen nämlich alle Daten seriell, also hintereinander, an die Display-Logik überwiesen werden. Schließlich läuft der Rasterstrahl, der das Bild zeichnet, auch seriell über den Bildschirm. Es gibt einfach keine Strahlenkanone, die ein ganzes Bild auf einmal anzeigen könnte. Dazu muß sich der Computer die einzelnen Daten aus dem Speicher holen. Das Aussehen des Bildes haben wir durch die Festlegung von Display-Window, Auflösung und Bitplane-Einhalten bereits festgelegt. Jetzt müssen die Daten noch aus dem Speicher geholt und an der richtigen Stelle des Bildschirms angezeigt werden. Diesen Vorgang nennt man auch »Data Fetching«.

Wie der Computer die Daten richtig aus dem Speicher holen soll, muß ihm natürlich mitgeteilt werden. Dazu muß die horizontale Startposition einer jeden Zeile in ein Register geschrieben werden. Und natürlich sollte der Computer auch wissen, wo er damit aufhören soll. Die sogenannten »Data Fetch Register« haben eine 4 Punkte große Auflösung. Jede Position ist also vier Pixels von der vorherigen entfernt. Pixel 0 ist Position 0; Pixel 4 ist Position 1 und so weiter.

Horizontales Scrolling und die Data Fetch Startposition sind wechselseitig voneinander abhängig. Es wird empfohlen, daß (aus Programmiergründen) die Data-Fetch-Werte auf eine Auflösung von 16 Pixels beschränkt werden. Die Hardware benötigt einige Zeit nach dem ersten »Holen« der Daten (also dem ersten Data Fetching), bevor diese Daten angezeigt werden können. Aus diesem Grunde besteht ein Unterschied zwischen dem Wert des Display-Window-Startes und dem Data Fetch Startwert. Der Unterschied entspricht im niedrigauflösenden Modus 8.5 Taktzyklen, im hochauflösenden Modus 4.5 Zyklen.

Der normale Wert des DDFSTART-Registers in LoRes-Modus liegt bei $0038, im HiRes-Modus bei $003C. Kluge Köpfe erkennen schon, daß die Hardwareauflösung von DIWSTART und DIWSTOP das Zweifache der Data-Fetch-Auflösung ist. So ergeben sich zum Beispiel auch die Normalwerte:

$81 / 2 - 8.5 = $38

$81 / 2 - 4.5 = $3C

Die Relation zwischen Start- und Endposition des Data Fetching ergibt sich aus:

DDFSTART = DDFSTOP - (8*(Wortzahl pro Zeile - 1) für LoRes

DDFSTART = DDFSTOP - (4*(Wortzahl pro Zeile - 2) für HiRes

Der Standard LoRes-Wert für DDFSTOP ist $00D0. Der Standard HiRes-Wert für DDFSTOP liegt bei $00D4. Auch diese Werte werden beim Anschalten durch die Software im Boot-ROM festgelegt und brauchen nur in eigenen Programmen mit Nicht-Standard-Werten neu gesetzt werden.

Dem System muß außerdem genau mitgeteilt werden, welche Bytes in jeder Zeile angezeigt werden. Um das zu tun, muß der sogenannte »Modulus« festgelegt werden. Der Modulus ist die Zahl der Bytes im Speicher, die zwischen dem letzten Speicherwort einer Zeile und dem ersten Wort der nächsten Zeile liegen. So werden die in linearer Form (von Startadresse bis Endadresse) gespeicherten Bitplane-Daten in »rechteckige« Form (also eine Zeile sequentieller Daten gefolgt von der nächsten Zeile) gebracht. Für ein einfaches Playfield, bei dem die Bitplanegröße der Display-Window-Größe entspricht, ist der Modulus 0, denn am Ende des letzten Datenwortes einer Zeile folgen gleich ohne Zwischenschritte die Daten für die nächste Zeile. Ist das Playfield breiter als der Teil, der angezeigt werden soll, ist der Modulus gleich der Anzahl der Pixels des Playfields abzüglich der Anzahl der Playfieldpixels, die angezeigt werden sollen. Mehr über größere Playfields aber in einem späteren Abschnitt.

In den meisten Computern wird dieser Wert automatisch durch Anschalten verschiedener Grafikmodi festgelegt. Beim Amiga wurde er veränderbar gemacht. Genau dadurch können wir Playfields darstellen, die größer als das Display-Window sind. Dazu aber später mehr im Kapitel über größere Playfields. Die folgende Grafik zeigt zunächst einmal, wie Data Fetch und Modulus die Speicherorganisation beeinflussen.

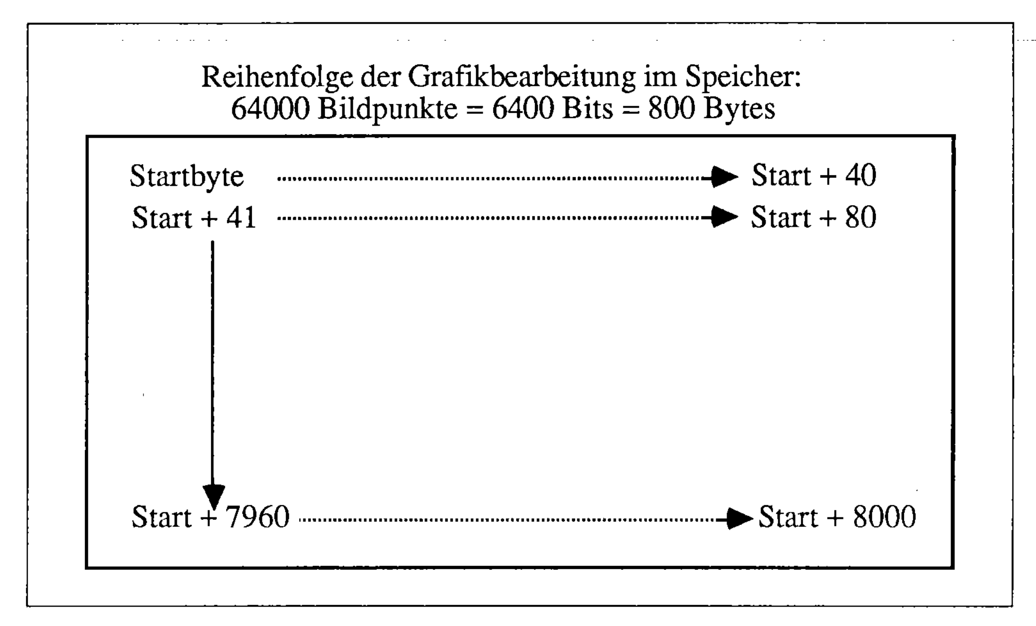


Bild 6.4: Die Funktion des Modulus bei der Bild-Anzeige

Die Bitplane-Adresszeiger BPLxPTH und BPLxPTL werden vom System benutzt, um die Daten durch das Data Fetching auf den Bildschirm zu bringen. Die Bitplane-Adreßregister sind dynamische Register, das heißt, sie ändern sich ständig. Der Wert, den wir zu Beginn festgelegt haben, ist nur der Anfangswert. Beginnt der Data-Fetch-Vorgang, werden die Adreßzeiger fortlaufend um ein Speicherwort inkrementiert. Das übernimmt automatisch die DMA-Kontrolle des Amiga, wenn wir sie angeschaltet haben. Nach jedem Inkrement wird das aktuelle Wort aus dem Bitplane-Adressregister geholt und angezeigt. Wenn die Bedingung des Linienendes erreicht ist (festgelegt durch DDFSTOP), wird der Modulus von der Hardware zu den Bitplanezeigern addiert, damit die nächste Zeile richtig angezeigt wird. Bei größeren Playfields wird also der Bereich zwischen DDFSTOP und dem durch den Modulus festgelegten neuen Zeilenanfang nicht angezeigt.

Das war wohl einer der schwierigsten Abschnitte, den Sie im Hardwareteil bisher gelesen haben. Sollten Sie es nicht verstanden haben, lesen Sie es bitte nochmal, denn Data Fetching und Modulus sind für das Verständnis der nachfolgenden Abschnitte wichtig.

Um Ihnen das Verständnis ein bißchen zu erleichtern, sehen wir uns noch einmal das Beispiel aus Bild 6.4 an. Nachdem hier die erste Linie angezeigt wurde, beinhalten die Bitplane-Adressregister BPLxPTH und BPLxPTL den Wert START+40 (Die letzte Anzeige war START+38; erst wird addiert, dann angezeigt und nicht etwa umgekehrt!). Der Modulus (in diesem Fall 4, weil die Playfieldgröße der Display-Window-Größe nicht entspricht) wird zum derzeitigen Wert (START+40) dazugezählt, nachdem durch DDFSTOP das Ende der derzeitigen Linie erkannt wurde. Die Daten, die in der nächsten Zeile erscheinen sollen, beginnen im Speicher also bei START+44. Bitte beachten Sie, daß die Adresszeiger immer eine gerade Nummer enthalten, weil die Grafikdaten nicht byteweise sondern wortweise gelesen und angezeigt werden (also jedesmal 16 Bit entsprechend zwei Bytes). Unser Beispiel hier bezieht sich auf die geringe Auflösung mit 40 Bytes pro Zeile. Wollen wir eine hochauflösende Grafik darstellen, müssen 80 Bytes pro Zeile ge«fetcht» (ausgelesen und angezeigt) werden.

Für den Modulus gibt es zwei Register: BPL1MOD für die Bitplanes mit ungerader Nummer, sowie BPL2MOD für die Bitplanes mit gerader Nummer.

Im Interlace-Modus funktioniert der Datafetch-Vorgang etwas anders. Hier müssen wir den Modulus neu definieren. Das ist deshalb nötig, weil wir es mit zwei separaten Rasterstrahldurchläufen für ein einziges Bild zu tun haben. Während des ersten Scanvorgangs werden die Linien ungerader Linienzahl auf den Bildschirm gebracht, und während des zweiten Scanvorgangs werden die Linien mit gerader Nummer angezeigt.

Die Bitplanes für ein Interlace-Display sind 400 Zeilen lang (hoch) und nicht 200. Angenommen, das Playfield im Speicher hat die normale Größe von 320 Pixels pro Zeile, dann beginnen die Daten für ein Interlace-Bild an den Byteadressen START (von uns festgelegter Wert) für die erste Zeile, die zweite Zeile beginnt bei START+40, START+80 ist der Beginn der dritten Zeile und so weiter.

Wir benutzen deshalb einen Modulus von 40, um die Linien des anderen Feldes zu überspringen. Dadurch können wir eine Interlace-Grafik im Speicher so anlegen, als hätten wir eine wirkliche Auflösung von 400 Punkten senkrecht. Die Interlace-Grafik ist also nicht so schwierig zu definieren wie beispielsweise an einem Apple IIc-

Computer. Bei diesem müssen nämlich wirklich zwei separate Bitplanes im Speicher definiert werden, so daß wir die erste Zeile in Bitplane 1, die zweite in Bitplane 2, die dritte wieder in Bitplane 1 und so weiter, setzen müßten. Am Amiga ist die Zusammensetzung der Grafik aus zwei Bitplanes zu einem Bild also reine Sache der Videologik; die Speicherorganisation hat nicht darunter zu leiden. So beginnt also für unsere (imaginären) Felder der Bitplanezeiger bei START für die ungeraden Felder, und bei START+40 für die geraden Felder.

Um die Behandlung der Bitplanezeiger für Interlace-Bilder zu regeln (sie müssen ständig bei jedem neuen Bildaufbau während des vertikalen Blanking zwischen den Werten der ungeraden und der geraden Felder gewechselt werden), wird der Copper benutzt. Informationen zum Copper folgen in einem der späteren Abschnitte.

6.1.6 Das Bild kommt auf den Bildschirm

All das, was wir bisher zur Playfield-Hardware gelernt haben, bringt das Bild aber noch lange nicht auf den Bildschirm. Wir haben lediglich alles für das Anzeigen des Bildes vorbereitet. Nachdem wir die Register gesetzt haben, starten wir durch das Setzen weiterer Register die Videohardware. Erst dann arbeiten die Custom-Chips ohne den Prozessor weiter.

Wir starten die Playfield-Anzeige, nachdem alle Register gesetzt sind, durch Anschalten der Bitplane-DMA. Das Anschalten des direkten Speicherzugriffs erfolgt durch Setzen des Bits BPLEN (BitPlane ENable) im Register DMACON (DMA-Control). Erst dann wird die Darstellung der Grafik in Bewegung gesetzt - das Data-Fetching läuft.

Jedesmal, wenn das Playfield wieder angezeigt werden soll, müssen allerdings die Bitplanezeiger wieder auf den Anfang des Bildspeichers zurückgesetzt werden. Dieses Zurücksetzen ist notwendig, weil die Zeiger durch das Data Fetching immer weiter inkrementiert werden. Am Ende des Playfieldspeichers angelangt, würde weiter inkrementiert und nur noch irgendwelche sinnlosen Speicherinhalte angezeigt werden. Zum Schluß des Playfields setzen wir also die Bitplanezeiger wieder auf den Anfang der Grafik (oder beispielsweise bei Aufwärtsscrolling an den Anfang der nächsten Zeile der

Grafik). Um dafür nicht auch noch Prozessorzeit zu brauchen, setzen wir wieder den Copper ein, da er selbst Befehle versteht, um Register zu ändern. Diese Befehle holt sich der Copper selbständig per DMA aus dem Speicher. Mehr dazu später in dem großen Abschnitt »Copper«. Das Anzeigen des Bildes per Data Fetch sowie das Zurücksetzen der Bitplanezeiger erfolgt im Normalfall durch die Betriebssystemsoftware. Das Betriebsystem baut automatisch die sogenannten Copperlisten auf; ist der Rasterstrahl am Ende des Bildes, erkennt der Copper das und kann die Bitplaneadreßzeiger wieder zurücksetzen, bevor der Rasterstrahl von neuem beginnt, das Bild von oben nach unten auf die Bildröhre zu zeichnen. Das Data-Fetching findet also jeweils vom Bildanfang links oben bis zum Bildende rechts unten statt und beginnt beim nächsten Bildaufbau (nach einer 60stel Sekunde) von vorn.

Schließlich und endlich wollen wir die Grafik auch noch über alle Ausgänge in Farbe ausgeben. Bei RGB passiert das automatisch; der Composite-Videoausgang benötigt jedoch ein sogenanntes Burst-Signal zur Farbdarstellung (egal ob PAL oder NTSC). Das Burst-Signal schalten wir ein durch Setzen von Bit 9 (Color enable) des Multifunktionsregisters BPLCON0.

6.1.7 Zusammenfassung für einfache Playfields

Wie Sie sehen, ist das Anzeigen eines einzigen, noch nicht einmal beweglichen Bildes schon recht kompliziert. Für viele lautet nun die Frage: Warum kann das alles nicht automatisch vor sich gehen, wie bei manchen anderen Computern? Darauf gibt es mehrere Antworten. Zunächst die erste: Dadurch, daß wir all diese teilweise unnötig erscheinenden Kleinigkeiten ändern können, haben wir mit dem Amiga eines der flexibelsten Grafiksysteme, die es überhaupt gibt. Es gibt eigentlich nichts, was sich an der Grafik des Amiga nicht ändern ließe. So ist dem Programmierer mehr Raum für seine Kreativität gelassen, auch wenn ihn das mehr Arbeit kostet als bei anderen Computern. Die zweite Antwort: So kompliziert ist die ganze Sache auch wieder nicht, weil uns bei der Anzeige des Bildes die Betriebssystemsoftware des Amiga die Hauptarbeit abnimmt. Nur, wenn Sie die Custom Chips direkt in Maschinensprache ansprechen wollen – sei es aus Abneigung gegen Betriebssysteme oder wegen der höheren

Flexibilität, die sie dadurch gewinnen – müssen Sie alle diese Arbeit selbst vornehmen.

Hier noch einmal alle Schritte, die Sie für die Darstellung eines einfachen Playfields benötigen, kurz zusammengefaßt:

a) Definition des Playfields

- Höhe in Zeilen und Breite in Pixels bestimmen.
- Bitplanes und Farben setzen:
  - Farbregister auf gewünschte Farbwerte setzen.
  - Bitplanes in gewünschten Speicherbereichen aufbauen.
  - Grafiken darin erstellen.
  - Bitplane-Register setzen:

    Anzahl der Bitplanes durch Bits 12-14 (BPU2 bis BPU0) in BPLCON0 bestimmen.

    Bitplane-Startpositionen durch BPLxPTH (als Longword) bestimmen.
- Auflösung bestimmen.
  - LoRes/HiRes-Wahl durch Bit 15 (HIRES) in BPLCON0.
  - Interlace-Modus an/aus: Bit 2 (LACE) in Register BPLCON0.

b) Speicher reservieren

- Notwendige Datenbytes nach folgender Formel kalkulieren (Bytes pro Linie) * (Linienanzahl des Playfields) * (Anzahl Bitplanes).

c) Anzuzeigenden Grafikbereich (Display Window) bestimmen.

- Startposition in DIWSTART schreiben.
  - Horizontale Position in Bits 0 bis 7
  - Vertikale Position in Bits 8 – 15
- Endposition in DIWSTOP schreiben.
  - Horizontale Position in Bits 0 bis 7
  - Vertikale Position in Bits 8 bis 15

d) Data-Fetching festlegen

- Register für DDFSTART und DDFSTOP beschreiben:
  - Startkoordinate in 4-Pixel-Auflösung in DDFSTART setzen.
  - Horizontale Endposition der Zeile in DDFSTOP schreiben.
  - Modulus in Registern BPL1MOD und BPL2MOD setzen.

e) Bild anzeigen

- Bitplane-DMA anschalten (Bit BPLEN in Register DMACON).
- Farbe für Composite-Video anschalten: Bit 9 in BPLCON0 setzen.
- Copperbefehlsliste schreiben und aktivieren.

6.2 Der Dual-Playfield-Modus

Durch Anschalten des Dual-Playfield-Modus ist es möglich, zwei Playfields gleichzeitig übereinandergelagert auf dem Bildschirm darzustellen. »Übereinander« heißt diesmal wirklich für das Auge sichtbar übereinander. Ein Playfield wird vor dem anderen gezeigt, wobei Teile des vorderen auch transparent sein können, um einen »Durchblick« auf das dahinter liegende Playfield zu gewähren. Dadurch hat man beim Design von Grafik wesentlich mehr Flexibilität; so kann man beispielsweise im Vordergrund das Kontrollpanel eines Panzers zeigen, während das dahinterliegende Playfield eine bewegte Grafik, sozusagen den Ausguck des Panzers, enthält. Zur Bewegung müssen wir dann nicht das gesamte Bild inklusive Panel ändern, sondern nur das Playfield, das die im Ausguck erscheinende Landschaftsgrafik enthält. Ein gutes Beispiel dafür ist das Amiga-Spiel »Arcticfox« von Electronic Arts. Wie so eine Anzeige aussieht und der Dual-Playfield-Modus funktioniert, sehen Sie auch in der folgenden Grafik.

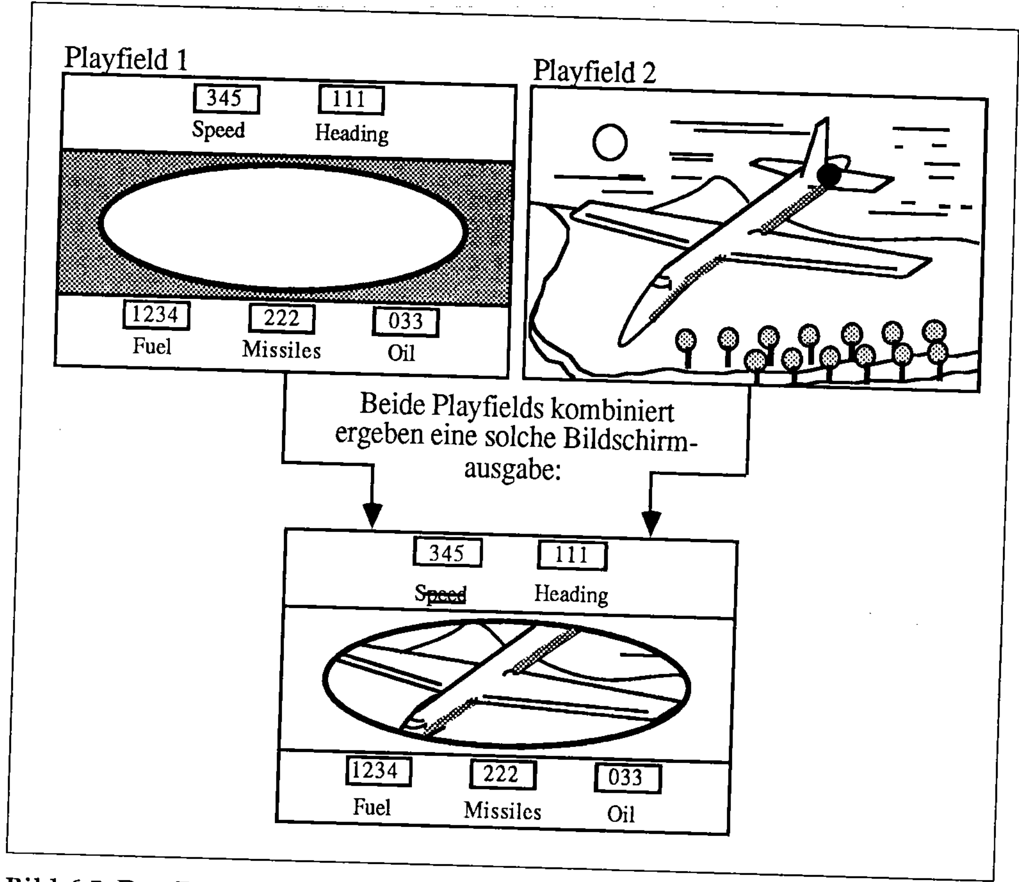


Bild 6.5: Der Dual-Playfield-Modus

Beide Playfields können unabhängig voneinander bewegt (gescrollt) werden. Eine Dual-Playfield-Grafik entspricht im großen und ganzen der eines einfachen Playfields, ist aber in einigen Aspekten anders zu betrachten: Jedes Playfield in einem Dual-Playfield-Display ist aus höchstens 3 Bitplanes zusammengesetzt. Die Farben in jedem Playfield (bis zu sieben plus Transparenz) werden aus unterschiedlichen Farbregistergruppen entnommen. Zum Aktivieren des Dual-Playfield-Modus muß ein Bit gesetzt werden.

Die obige Abbildung zeigt genau, daß eine der Farben im Playfield transparent, also durchsichtig, ist. Bei Playfield 1 wird die Hintergrundfarbe 0 durchsichtig, bei Playfield 2 ist Farbe 8 durchsichtig. Diese Transparenz ermöglicht erst den sogenannten »Fenster-Effekt«.

Bei Playfields, die mit unterschiedlicher Geschwindigkeit über den Bildschirm scrollen, erscheint durch die Transparenz ein 3D-Effekt, als wenn man über ein bestimmtes Terrain fliegen würde. Das untere Playfield macht den Eindruck, als wenn es weiter entfernt liegen würde als das obere Playfield. Der Dual-Playfield-Modus ermöglicht also besonders Programmierern von Spielen ungeahnte Effekte. Doch auch für CAD-Anwendungen ist diese Technik interessant (zum Beispiel durch Übereinanderlegen von Grundriß eines Hauses und elektrischer Verkabelung).

6.2.1 Die Organisation der Grafik im Dual-Playfield-Modus

Die drei Bitplanes mit ungeraden Plane-Nummern (Plane 1, 3 und 5) werden durch die Hardware zu Playfield 1 zusammengruppiert. Die drei geradzahligen Bitplanes (2, 4 und 6) werden zu Playfield 2 gruppiert. Die Bitplanes werden einander zugeordnet wie im folgenden Bild gezeigt wird. Beachten Sie bitte, daß im HiRes-Modus aus Timing-Gründen höchstens zwei Bitplanes pro Playfield benutzt werden dürfen – Bitplanes 1 und 3 für Playfield 1 und Bitplanes 2 und 4 für Playfield 2. Die Playfields können voreinander oder hintereinander angezeigt werden, indem ein sogenanntes «Swap-Bit» an- oder ausgeschaltet wird.

Zahl der angeschalteten Bitplanes	Playfield 1	Playfield 2
0	keine	keine
1	1	
2	1	2
3	1, 3	2, 4
4	1, 3	2, 4
5	1, 3, 5	2, 4, 6
6	1, 3, 5	2, 4, 6

Bild 6.6: Die Zuordnung von Bitplanes zu Dual-Playfields

6.2.2 Farbauswahl im Dual-Playfield-Modus

Die Hardware des Amiga interpretiert – bei angeschaltetem Dual-Playfield-Modus – die Farbregisternummern aus den Bitkombinationen von Bitplanes 1, 3 und 5. Das Bit aus Plane 5 stellt das höchstwertige Bit dar, Plane 0 das niedrigstwertige Bit. Diese Bitkombinationen greifen auf die ersten 8 Farbregister wie folgt zu. Bei Playfield 1 ergibt die Kombination der drei benutzten Planes einen Wert von binär 000 bis 111 (also dezimal von 0 bis 7). 000 gibt Transparenz an, die restlichen Werte entsprechen der Nummer des Farbregisters.

Bei Playfield 2 wird dieselbe Verfahrensweise angewendet. Hier wird das Farbregister durch die Planes 2, 4 und 6 angewählt, wobei Plane 6 das höchstwertige Bit darstellt, Plane 2 das niedrigstwertige. Die so zustandekommenden Bitkombinationen wählen die Farbe aus den nachfolgenden Farbregistern aus. Dabei gilt: Dezimalwert der Bitkombination + 8 = Registernummer, also beispielsweise 000 für Transparenz, 001 für COLOR9 bis 111 für das Register COLOR15. Wo immer Farbe 0 angewählt ist, erscheint die Farbe desjenigen Objektes, das »hinter« dem Playfield ist (also ein anderes Playfield, ein Sprite, oder, sollte gar nichts dahinter liegen, die in COLOR0 stehende Farbe als Hintergrundfarbe).

Wie bereits gesagt, können im HiRes-Modus höchstens zwei Bitplanes pro Playfield verwendet werden. Daraus ergeben sich Kombinationsmöglichkeiten von 00 bis 11 für jedes Playfield. Die Berechnung des richtigen Farbregisters ist identisch zum LoRes-Modus, also 00 für Transparenz, 01 für Farbregister COLOR1 beim ersten Playfield oder 01 für Farbregister COLOR9 beim zweiten Playfield. Das geht bis zur Kombination 11 für COLOR3 beziehungsweise COLOR11.

6.2.3 Was kommt vorne, was steht hinten

Welches der zwei Playfields vorn und welches hinten liegt, wird durch die »Priorität« der Playfields festgelegt. Dasjenige Playfield, das Priorität hat, liegt vor dem anderen. Wird nichts festgelegt, hat automatisch Playfield 1 Priorität vor Playfield 2. Wollen wir Playfield 2 im Vordergrund anzeigen, muß Bit 6 des Registers

BPLCON2 gesetzt werden. Dieses Kontrollbit hat den Namen PF2PRI (für PlayField 2 PRIority).

Die Priorität zwischen Sprites und Playfields kann ebenfalls geändert werden. So kann beispielsweise ein Sprite »zwischen« den zwei Playfields angezeigt werden. Näheres dazu im Abschnitt »Kontrollhardware«.

Zwei Playfields können unabhängig voneinander kontrolliert werden, sie können beispielsweise unterschiedliche Größe im Speicher und auf dem Bildschirm haben und auch unabhängig voneinander gescrollt werden. Beim Scrolling ist allerdings etwas zu beachten, was die »Unabhängigkeit« der zwei Playfields wieder ein wenig einschränkt. Wenn ein Playfield gescrollt und das andere unbeweglich bleiben soll, ist besondere Vorsicht geboten, damit ein scrollendes Bild nicht das andere »mitreißt«. Werden Low Resolution-Playfields bewegt, muß der Fetch-Vorgang ein Speicherwort mehr umfassen, als die Breite des zu scrollenden Playfields umfaßt. In HiRes-Modus gilt dasselbe für zwei Speicherworte. Dies ergibt sich deshalb, weil während des Scrollens Daten dargestellt werden müssen, aber nur ein DDFSTART- und ein DDFSTOP-Register existieren. Diese Register werden von beiden Playfields geteilt. Wenn nun ein Playfield gescrollt werden soll, das andere dabei aber still stehen soll, müssen noch immer die Datafetch-Start- und -Stopregister für das Scrolling des Playfields angepaßt werden. Deshalb müssen der Modulus und die Bitplane-Adreßregister für das Playfield angepaßt werden, das stehenbleiben soll, damit es nicht mitscrollt, sondern seine Position beibehält. In LoRes-Modus werden dabei die Zeiger und der Modulus um -2 pro Fetch-Inkrement angepaßt. In HiRes ändert sich dieser Anpassungswert auf -4. Mehr zum Thema Scrolling aber im übernächsten Abschnitt.

6.2.4 Arbeiten mit dem Dual-Playfield-Modus

Wir können, wie bei allen DMA-abhängigen Hardwareteilen, den Dual-Playfield-Modus erst dann nutzen, wenn wir ihn aktiviert haben. Dazu ist wieder eines der Bits des BPLCON0-Registers zuständig. Bit 10, genannt DBLPF, aktiviert diesen Modus, indem es auf 1 gesetzt wird. Wird der Inhalt auf 0 gesetzt, wird der Dual-Playfield-Modus wieder ausgeschaltet.

Durch das Setzen des DBLPF-Bits ändert sich für die Hardware die Art, in der die Bitplanes zur Farbinterpretation gruppiert werden (ungerade Bitplane-Nummern zu Playfield 1, gerade Bitplane-Nummern zu Playfield 2) und die Art, wie die Hardware die Bitplanes auf dem Bildschirm bewegt.

6.2.5 Zusammenfassung: Erstellung eines Dual-Playfield-Displays

Die Schritte zur Erstellung von Dual-Playfields entsprechen im großen und ganzen denen, die auch zur Erstellung einfacher Playfields benutzt werden. Die sich davon unterscheidenden Punkte betreffen das Setzen der Farbregister, die Gruppierung der Bitplanes zu Playfields sowie das Setzen von zwei Modulus-Registern statt einem. Die Farbregister 0 bis 7 sind für das erste Playfield reserviert, das sich aus Planes 1, 3 und 5 zusammensetzt. Farbregister 8 bis 15 gehören dem zweiten Playfield und damit den Bitplanes 2, 4 und 6 zu. Der Modulus muß separat in BPL1MOD und BPL2MOD für jedes Playfield geregelt werden. Mit dem Bit PF2PRI kann optional die Priorität der Playfields geändert werden. Das Aktivieren des Dual-Playfield-Modus geschieht schließlich mit Bit 10 (DBLPF) in Register BPLCON0 (muß auf 1 gesetzt werden).

6.3 Playfieldgrafiken beliebiger Größe

Bisher haben wir nur Playfields besprochen, deren Größe identisch mit der Größe des Display-Windows ist. Es geht jedoch auch kleiner oder größer. Wie es kleiner geht, haben wir bereits besprochen: Wir verkleinern einfach den Bereich der Grafik, ändern entsprechend die Datafetch-Registerinhalte und passen das Display-Window auf die kleinere Größe an, damit nicht um unsere Grafik herum irgendwelcher »Datenmüll« erscheint.

Größere Bilder sind etwas komplizierter. Der folgende Abschnitt soll Ihnen zeigen, wie die Hardware mit Playfields umgeht, deren Größe und Speicherbedarf über die Größe des sichtbaren Bereiches hinausgehen, wie Display-Windows größer oder kleiner als normale Playfield-Größe werden können und wie man das Display-Window in einem großen Bild bewegt (Lupen- oder Guckloch-Effekt).

6.3.1 Wie eine »Riesengrafik« aufgebaut ist

Wenn wir ein Bild definieren, das größer als das Display-Window ist, müssen wir auswählen, welchen Teil der Grafik wir im Display-Window anzeigen. Das Anzeigen eines Teiles von großen Grafiken unterscheidet sich nur in wenigen Punkten von den bisher beschriebenen Techniken: Der Modulus muß für ein großes Bild neu definiert werden, wie bereits oben beschrieben. Und für die Grafik muß natürlich mehr Speicher reserviert werden, da sie entsprechend größer ist.

Wie hängt nun der Modulus mit der Größe der Grafik zusammen? Sehen Sie sich dazu einmal das folgende Bild an.

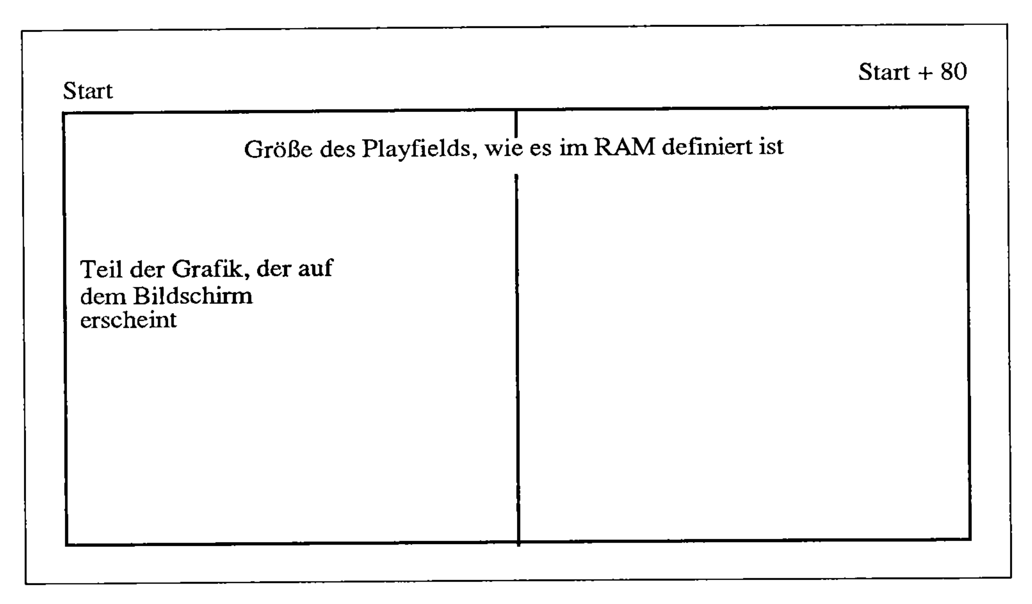


Bild 6.7: Ein doppelt breites Playfield

Die hier gezeigte Grafik ist doppelt so breit wie der Bereich, der angezeigt wird. Anzeigen wollen wir nur die linke Hälfte. Wir legen die Grafik im Speicher einfach so an, als hätten wir eine Bildschirmauflösung, die der Größe des Playfields entspricht. Leider hat der Bildschirm nun nicht dieselbe Auflösung wie die Grafik, sondern nur die Hälfte davon. Im Normalfall würde der Computer bei einer Auflösung von 320 Punkten pro Zeile beim vierzigsten Byte die Zeile

wechseln. So würden Zeilen, die in unserer Grafik eigentlich rechts im nicht sichtbaren Teil liegen, mit eingeschoben werden. Das gäbe ein schlechtes Aussehen, weil immer erst eine 40-Byte lange Zeile des linken Teils erscheint, dann als zweite Zeile die 40 Bytes, die eigentlich rechts liegen sollten, dann wieder eine 40 Byte-Zeile, die links liegt, usw. Um das zu vermeiden, verwenden wir nun den bereits erwähnten Modulus.

Wir erinnern uns, daß die Darstellung der Grafik durch den sogenannten Data-Fetch-Vorgang geschieht. Unsere Grafik ist nun 80 Bytes breit. Davon sollen aber immer nur die linken 40 Bytes angezeigt werden. Weil nun von jeder Zeile immer nur 40 Bytes ge»fetcht«, also aus dem Speicher geholt und angezeigt werden sollen, müssen nach dem Anzeigen einer jeden 40 Byte großen Zeile 40 Bytes ausgelassen werden. Diesen Wert definieren wir durch den Modulus.

Was tun wir, wenn wir die rechte Hälfte des Bildes anzeigen wollen? Wir wissen, daß vor dem Aufbau eines jeden neuen Bildes, also nach einem vollständigen Rasterstrahldurchlauf von links oben nach rechts unten, ein sogenanntes Blanking stattfindet. Der Amiga ermöglicht es, nach jedem Rasterstrahldurchlauf während des Blankings (bevor das Bild wieder angezeigt wird), mit Hilfe des Coppers eine sogenannte »Vertical Blanking Interrupt Routine« anzuspringen. Nach jedem Rasterstrahldurchlauf erfolgt automatisch ein sogenannter »Interrupt«, eine Unterbrechung durch ein Hardwaresignal. Durch dieses Signal können wir in die eben genannte Routine springen, die aus einer Reihe von Copper-Befehlen bestehen kann. Im Normalfall befindet sich in dieser Befehlsliste das Zurücksetzen der Bitplane-Adreßzeiger auf den Anfang des Bildes.

Wollen wir nun die rechte Hälfte anzeigen, ändern wir einfach die Interrupt-Routine, um in die Bitplane-Adreßzeiger den Wert START+40 als neuen Startwert zu schreiben. Der Modulus bleibt bei 40, da auch hier vierzig Bytes übersprungen werden müssen. In diesem Fall sind es die vierzig Bytes, die links liegen und »verschwinden« sollen. Am Ende der Zeile angekommen, enthalten die Bitplanezeiger den Wert START+80. Vor dem Anzeigen der nächsten Zeile wird zur Adresse in den Bitplanezeigern der Modulus (40) addiert. Damit sind wir bei START+120 und somit auch wieder in der rechten Hälfte des Bildes.

Wollen wir die Mitte des Bildes oder irgend einen anderen Teil dazwischen in das Display-Window bringen, ändern wir die Rücksetzung der Bitplanezeiger in der Vertical Blanking Routine auf einen entsprechenden Wert. Mehr zum Blanking aber im Abschnitt über den Copper.

Beachten Sie, daß im hochauflösenden Modus zweimal so viele Bytes wie im niedrigauflösenden Modus ge»fetcht« werden müssen. Entsprechend muß auch der Modulus geändert werden.

Bei großen Playfields werden die Datafetch-Register auf dieselbe Art und Weise gesetzt, wie auch bei den einfachen Playfields. Der Speicherbedarf ist größer, wird aber mit derselben Formel berechnet wie bei einfachen Playfields:

Bedarf = Bytes/Linie * Linien des Playfields * Anzahl der Bitplanes.

Das Playfield aus unserem Beispiel benötigt also (80 * 200 * Anzahl der Bitplanes) Bytes Speicher. Angenommen, wir wollen nur vier Farben, also zwei Bitplanes verwenden, benötigen wir damit 32 000 Bytes für ein LoRes-Non-Interlace-Bild, das die doppelte Größe eines normalen LoRes-Bildes hat.

All diese Aussagen beziehen sich auf Bilder, die breiter als das Display-Window sind. Bilder, die länger (höher) sind, werden genauso behandelt, wie einfache Playfields. Wollen wir einen anderen Teil einer langen Grafik sehen, ändern wir beim vertikalen Blanking den Adresszeiger entsprechend. Wollen wir beispielsweise die oberen drei Zeilen einer LoRes-Grafik nicht sehen, zählen wir zum Bitplanezeiger einfach drei mal 40 Bytes dazu (drei Zeilen), und schon wird der Teil der Grafik dargestellt, der etwas weiter »unten« liegt. Ist unsere Grafik sowohl in der Länge als auch der Breite größer, wird einfach das Wissen aus diesem Absatz und dem vorherigen kombiniert.

6.3.2 Änderungen am Display-Window

Das Display-Window bildet ein Rechteck. Die linke obere Ecke ist durch die horizontalen und vertikalen Lo-Res-Non-Interlace-Koordinaten HSTART und VSTART im Register DIWSTART festgelegt. Damit ist der HSTART-Wert auf die ersten 256 Positionen

rechts von dem kleinstmöglichen linken Punkt beschränkt. Die 8 Bits, die den VSTART-Wert angeben, sind auf die ersten 256 Positionen, vom obersten Punkt des Displays aus gezählt, beschränkt. Beim Interlace-Modus sollte man beachten, daß eine gerade Zahl für die Anzahl der Linien des Display-Windows verwendet wird, damit die beiden zu einem Bild zusammengesetzten Playfields gleich groß sein können.

Die rechte untere Ecke des Display-Windows ist auf die rechte untere Hälfte des Bildschirms beschränkt. Die Stop-Position wird durch die 8-Bit-Werte HSTOP und VSTOP im Register DIWSTOP festgelegt. Wie diese Werte den Bildschirmkoordinaten entsprechen, haben wir bereits im Abschnitt »einfache Playfields« besprochen.

Die maximale Größe eines Displays-Windows, in dem Grafik dargestellt werden kann, wird bestimmt durch die Maximalanzahl von Zeilen und Spalten. Wir haben gesagt, daß wir auch Display-Windows öffnen können, die größer sind als der Normalwert, so daß wir quasi das gesamte Monitorbild abdecken können. Das stimmt nicht ganz. Vertikal gesehen sind die Beschränkungen einfach: Bevor das Bild wieder gezeichnet wird, ist das sogenannte Vertical Blanking aktiv. Da der Computer beim Blanking auch die «Vertical Blanking Interrupt Routine« durchlaufen muß, benötigt er etwas Zeit; in dieser Zeit zeichnet der Rasterstrahl nur die Hintergrundfarbe. Dieser Bereich liegt von Zeile 0 bis Zeile 19, insgesamt also 20 Zeilen. Das läßt uns 242 Zeilen Bildschirm, in die wir Grafik setzen können (beim PAL-Amiga sind es 256); im Interlace-Modus sind es 484 Zeilen (beim PAL-Amiga 512).

Horizontal sieht die Situation ähnlich aus. Die Hardware setzt uns ein Limit der darstellbaren Grafik, da sich der Datafetch-Vorgang nach rechts allerhöchstens bis zum Wert DDFSTOP $D8 erstrecken kann und links spätestens bei $18 beginnt. Das gibt uns im LoRes-Modus eine maximale Anzahl von 25 darstellbaren Worten pro Zeile. In hoher Auflösung ist der Maximalwert nicht das doppelte, sondern nur 49 Worte, da die Rechtsbeschränkung bei $D8 bleibt. Ein weiteres Limit wird uns durch das horizontale Blanking gesetzt (nachdem ein Rasterstrahl rechts angekommen ist, benötigt er etwas Zeit, wieder an den Anfang der nächsten Zeile zu gelangen. In dieser Zeit rechnet die Hardware auch den Modulus zum Bitplanezeiger hinzu). Dieses Blanking beschränkt uns auf allerhöchstens 376 Pixels (LoRes-Größe)

pro Zeile (23,5 Datenworte). In hoher Auflösung verdoppelt sich dieser Wert. Diese horizontale Beschränkung verhält sich bei PAL-Geräten und NTSC-Geräten identisch. Zusätzlich sollte man beachten, daß ein Datafetch-Startwert links von $38 aus Timing-Gründen einige Sprites abschalten wird. Darum wird eine Vergrößerung des Display-Windows nach links nur empfohlen, wenn keine Sprites im Bild dargestellt werden sollen.

Damit wäre das Vergrößern und Verkleinern der darstellbaren Grafik abgehandelt. Wie verschieben wir nun das Display-Window über den Bildschirm? Wollen wir das Display-Window über der Grafik hin- und herschieben, ändern wir die Werte für DIWSTART und DIWSTOP. Bei der Bewegung muß jedoch darauf geachtet werden, daß die Fetch-Werte DDFSTART und DDFSTOP auch angepaßt werden. Das ist deshalb nötig, weil wir sonst die Grafik (zumindest erscheint es so) mitverschieben würden. Wird der Data-Fetch-Wert angepaßt, erwecken wir beim Bewegen eines kleinen Display-Windows den Eindruck, als wenn wir mit einer Lupe über die Grafik fahren würden (wobei der Bereich um die Lupe herum verdeckt ist, weil außerhalb des Display-Windows nur die Hintergrundfarbe dargestellt wird).

6.4 Playfield-Scrolling

Sicher haben Sie schon des öfteren bewegliche Hintergrundgrafiken gesehen, die von oben nach unten, von links nach rechts, vielleicht auch diagonal wandern. Diese Art von Bewegung nennt man Scrolling. Ein gutes Beispiel ist das Spiel »Marble Madness«, das von einem Spielhallenautomaten auf den Amiga umgesetzt wurde.

Vertikales Scrolling ist recht einfach: Bei jedem vertikalen Blanking-Vorgang ändern wir einfach die Bitplane-Adresszeiger auf die nachfolgende oder vorhergehende Zeile der Grafik; je nach Richtung des Scrolling.

Horizontales Scrolling arbeitet komplizierter. Um von links nach rechts oder rechts nach links zu scrollen, muß der Data-Fetch-Vorgang um ein Wort erweitert werden. Das Anzeigen dieses Wortes muß dann entsprechend der Scroll-Geschwindigkeit einer Verzögerung unterliegen. Beide Typen von Scrolling haben aber gemeinsam, daß

die Bitplane-Adresszeiger während des vertikalen Blanking geändert werden müssen.

6.4.1 Vertikale Bewegung des Playfields

Wir wissen, daß ein Bild von links oben nach rechts unten durch das Fetching angezeigt wird. Ist das Bild fertig gezeichnet, ist der Rasterstrahl damit am letzten Punkt des Bildes. Dadurch wird ein Signal ausgelöst, ein sogenannter Interrupt, der dafür sorgt, daß eine Reihe von Befehlen ausgeführt werden. In dieser Befehlsliste wird normalerweise der Bitplane-Adreßzeiger (BPLxPTH und BPLxPTL) auf den Anfang (STARTadresse) zurückgesetzt. Innerhalb dieser Interrupt-Routine fügen wir nun Befehle ein, die die Bitplanezeiger auf einen dem Scrolling entsprechenden anderen Wert setzen. Angenommen, wir wollen ein langes Playfield von unten nach oben scrollen. Im Vertical Blanking zählen wir dann zum ursprünglichen Wert START, bei dem das Data-Fetching beginnt, den Bytewert, der der Anzahl der Pixels pro Zeile entspricht, dazu. Bei einem LoRes-Bild, müssen also 320/8 = 40 Bytes dazugezählt werden. Damit müssen die Bitplanezeiger auf den Wert START+40 gesetzt werden. Ist das Bild danach wieder einmal vollständig ge»fetcht«, wiederholt sich der Vorgang. Damit müssen wir im vertikalen Blanking die Bitplanezeiger auf START+80 setzen, beim nächsten Mal auf START+120, und so weiter.

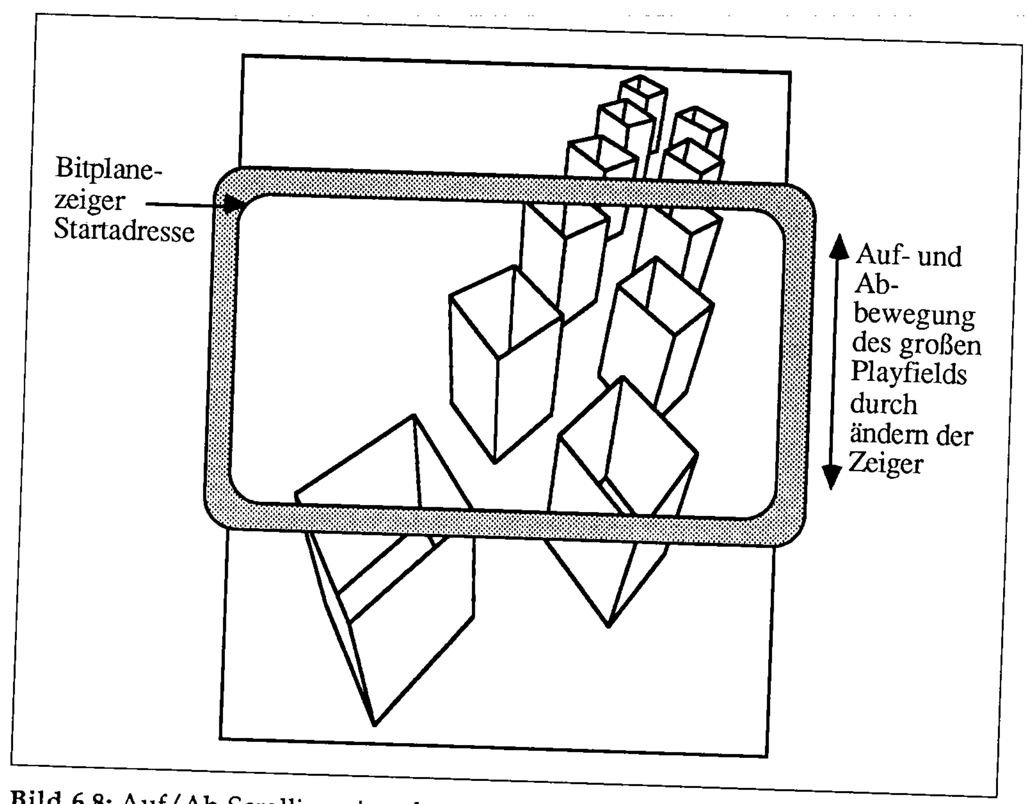


Bild 6.8: Auf/Ab-Scrolling eines langen Playfields

Zusammengefaßt muß man beim Erstellen eines Auf/Abscrolling folgendes beachten:

- Es muß Software geschrieben werden, die die nach jedem Bildaufbau neu zu setzenden Adressen der Bitplanezeiger entsprechend dem gewünschten Scrolling berechnet. Die so berechneten Werte müssen an die Copperliste übergeben werden (siehe Abschnitt »Copper«).

- Die Grafikdaten müssen ausreichend vorhanden sein. Scrollen wir weiter als die Grafikdaten reichen, erscheint Müll auf dem Bildschirm.

Der kleinstmögliche Schritt bei vertikalem Scrolling ist ein Zweihundertstel (also eine Zeile) des Bildes. Wird ein kleinerer Wert genommen, scrollt nichts, sondern das Playfield bewegt sich unkontrolliert über den Bildschirm. Natürlich kann man auch immer zwei Zeilen weiterrücken; dadurch wird das Scrolling schneller. Je mehr

Zeilen pro Blanking gescrollt werden, desto schneller aber auch desto ruckartiger bewegt sich das Bild.

Soll im Interlace-Modus gescrollt werden, kann bis zu ein Vierhundertstel (also wieder 1 Zeile) des Bildes gescrollt werden. Da die Videologik mit zwei Bitplanes arbeiten muß, kann es dabei zu Unregelmäßigkeiten im Erscheinungsbild kommen (noch stärkeres Flimmern, Verschiebungen der ungeraden Bitplanes gegenüber den geraden und so weiter). Das kann zwar durch cleveres Timing beim Setzen der Zeiger ausgeglichen werden, ist jedoch viel zu umständlich. Es wird deshalb empfohlen, im Interlace-Modus immer zwei Zeilen auf einmal zu scrollen, um das Platzverhältnis der geraden und ungeraden Felder gleich zu halten.

6.4.2 Links/Rechts-Scrolling

Ein Playfield kann von links nach rechts oder von rechts nach links gescrollt werden. Die Geschwindigkeit des Scrolling wird kontrolliert durch die Angabe der Verzögerung des Fetching (Angabe in Pixels). Verzögerung bedeutet, daß ein zusätzliches Datenwort vom Speicher auf den Bildschirm gebracht wird, jedoch nicht sofort angezeigt wird.

Nun ein Beispiel für Scrolling nach rechts: Das zusätzliche Wort wird links vom sichtbaren Teil des Bildes plaziert, also kurz vor dem normalen DDFSTART-Wert. Wenn das Display sich nach rechts bewegt, erscheinen die zusätzlichen Daten des linken Wortes auf der linken Seite des Display-Windows, während die Daten des rechts liegenden Wortes nach rechts verschwinden. Für jedes angegebene Pixel Verzögerung (Fachbegriff DELAY) bewegt sich das Bild um einen Punkt nach rechts. Je größer der Wert DELAY gewählt wird, desto größer ist die Geschwindigkeit des Scrolling. Es ist ein Delay bis zu 15 Pixel möglich. Im HiRes-Modus wird das Bild in 2-Pixel-Schritten bewegt. Der Verzögerungsvorgang bezieht sich also auf LoRes-Pixel.

Die Funktionsweise des Delay zeigt das folgende Bild. Die Verzögerung wird angeschaltet durch Setzen der Delay-Bits im Register BPLCON1. Es handelt sich dabei um Bits 7 bis 0. Die Bits 3-0 geben den Wert PF1H (PlayField 1 Horizontal) für die Bitplanes ungerader Zahl ab, Bits 7 bis 4 ergeben den Wert PF2H für die

Verzögerung der Bitplanes gerader Zahl. Für Single-Playfield-Betrieb müssen PF1H und PF2H simultan auf den gewünschten Wert (0 bis 15) gesetzt werden, bei Dual-Playfield-Modus müssen die Werte getrennt für die einzelnen Playfields gesetzt werden.

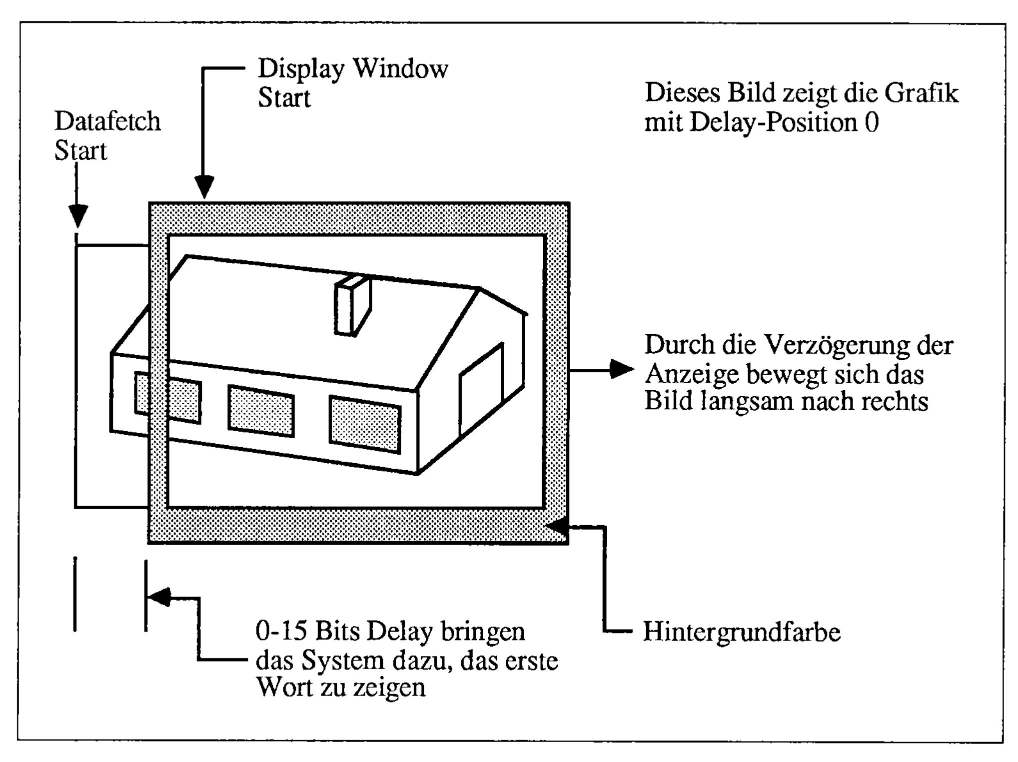


Bild 6.9: Horizontales Scrolling durch Verzögerung des Datafetching

Durch das Setzen der Delay-Bits bewegt sich das Bild automatisch nach rechts, und zwar innerhalb der Anzeige eines ganzen Bildes (1 Rasterstrahldurchlauf) um die Anzahl von Pixels, die der Delay-Wert bestimmt.

Wie bereits gesagt, muß das Fetching ein Datenwort mehr umfassen. Der Normalwert für nicht scrollende Playfields liegt bei $38, wir müssen also, soll die Display-Window-Größe nicht verändert werden, einen Wert von $30 für DDFSTART einsetzen. Das wird allerdings Sprite 7 abschalten. Man sollte also bei Programmierung von Animation beachten, daß Sprite 7 nicht gleichzeitig mit horizontal scrollenden Playfields benutzt wird. Der DDFSTOP-Wert bleibt un-

verändert. Bitte beachten Sie, daß sich die Data-Fetch-Änderungen auf beide Playfields beziehen, wenn Dual-Playfield-Modus aktiv ist.

Das erste Wort, das in einer Zeile angezeigt werden soll, ist in diesem Fall das zusätzliche Datenwort (auch wenn es zu Anfang noch nicht angezeigt wird, sondern erst langsam sichtbar wird).

Als Beispiel für ein horizontales Scrolling nehmen wir wieder unser Playfield aus dem Abschnitt über große Playfields, also ein 80 Bytes breites Playfield mit 40 anzuzeigenden Bytes. Da das horizontale Scrolling das zusätzliche Datafetch-Wort benötigt, sind die Daten für jede Zeile damit 42 Bytes lang. Nach dem Anzeigen der ersten Zeile beinhalten die Bitplane-Adreßzeiger den Wert START+42. Da eine Zeile 42 Bytes (und nicht mehr 40) breit ist, gibt ein Modulus von 38 den korrekten Startpunkt der nachfolgenden Linie. Durch die Addition des Modulus erreichen die Bitplane-Adreßzeiger also den Wert START+80. Die nachfolgende Zeile wird aus dem Speicher geholt und angezeigt, von START+80 bis START+120. Dann wird wieder der Modulus addiert, und so weiter.

Die Animation von Playfields ist auch mit Hilfe der Animations-Bibliotheken des Betriebssystems möglich. Die Betriebssystemroutinen sind etwas einfacher zu benutzen als das direkte Ansprechen der Hardware; sie gehen jedoch bei der Behandlung des Playfieldscrolling genauso vor, wie hier beschrieben.

6.5 Hold & Modify - 4096 Farben gleichzeitig

Der Hold & Modify-Modus erlaubt es, mehr als 32 Farben gleichzeitig anzuzeigen. Der Hold & Modify-Modus kann nicht mit dem Aufbau der bisherigen Playfields verglichen werden, ist jedoch trotzdem ein Bestandteil der Playfield-Hardware. Die Anzeige von Farbe basiert bei normalen Playfields auf dem Prinzip, daß beim Datafetch die Bitkombination der übereinandergelegten Bitplanes als Farbregisternummer interpretiert wird, und die Farblogik (Color output curcuit) mit dem Inhalt dieses Registers «gefüttert» wird. Pro Datafetch passiert das 16mal, einmal für jeden Bildpunkt.

Im Hold & Modify funktioniert das Prinzip der Farbauswahl etwas anders: Hier wird nicht bei jedem Punkt der Farbwert neu ausgewählt. Der Wert, der in der Farblogik für den letzten Punkt aktuell war, wird gespeichert und anschließend um eine Farbkomponente modifiziert. Erst nach der Änderung des Rot-, Grün- oder Blauwerts, wird das Pixel in der richtigen Farbe auf den Bildschirm geschrieben. Die Methode wird durch ihren Namen ausreichend beschrieben: »Hold« für das Halten des Farbwertes und »Modify« für die Änderung um die Farbkomponente.

Im folgenden wollen wir Hold & Modify nur noch durch die Abkürzung HAM umschreiben. Im HAM-Modus werden sechs Bitplanes genutzt. Planes 5 und 6 übernehmen die Modifikationsfunktion: Sie sagen aus, in welcher Art und Weise die Bits aus den Planes 1 bis 4 zu behandeln sind (als Farbwert oder als Rot-, Grün- oder Blaumodifikationswert). Das funktioniert folgendermaßen:

- Ist die Kombination der Planes 5 und 6 gleich 00, nimmt die Hardware eine »normale« Farbwahl vor. Auf diese Weise gibt der Wert von Planes 1 bis 4 einen Wert von 0 bis 15, der das entsprechende Farbregister anspricht (alle Register ab 16 können nicht verwendet werden).

- Ist die Kombination 5-6 gleich 01, wird die Farbe des unmittelbar links liegenden Punktes in der Farblogik gehalten und der Blauwert geändert. Die Änderung des Blauwerts geschieht durch Planes 1 bis 4. Sie ersetzen die vier »Blau«-Bits des Inhalts der Videologik.

- Eine 5-6-Kombination von 10 hält den Farbwert des unmittelbar links liegenden Punktes und ersetzt die Rot-Intensität der Farbe durch die Kombination aus Planes 1 bis 4.

- Wenn die Kombination aus Planes 5 und 6 den Wert 11 hat, wird die Farbe des unmittelbar links liegenden Pixels gehalten und um die Grün-Bits geändert. Die Ersetzung der vier »Grün«-Bits entspricht der Kombination aus Planes 4 bis 1.

Werden nur 5 Bitplanes benutzt, während HAM aktiv ist, nimmt die Videologik für das Bit aus der sechsten Bitplane automatisch einen Wert von Null an. Damit kann nur eine Auswahl zwischen den beiden erstgenannten Punkten erfolgen: Entweder der Farbwert wird aus den Registern 0 bis 15 gewählt, oder der Blau-Anteil ändert sich. Das klingt zwar nicht sehr sinnvoll, ist es aber durchaus in einigen

Fällen. Beispiel: Wir haben eine Grafik, die außer einer schattierten blauen Blumenvase nur wenig mehr als 16 Farben benötigt. Die Schattierung der Vase nehmen wir nun mit den Blauwerten per HAM-Methode vor. Damit haben wir alle möglichen Blauschattierungen abgedeckt und können sogar noch verschiedene Violett-Schattierungen für eine zweite Vase auf den Bildschirm zaubern. Trotzdem haben wir uns 8000 Bytes Speicherplatz gespart, da die sechste Bitplane nicht benutzt ist.

Im HAM-Modus kann es sogar genügen, nur ein einziges Farbregister (COLOR0, die Hintergrundfarbe) zu verwenden. Der Rest des Bildschirms wird als reine Modifikation der Hintergrundfarbe angesehen. Trotzdem bringt man damit eine erstaunlich große Anzahl von Farben zustande.

Der HAM-Modus wird angeschaltet durch Bit 11 des BPLCON0-Registers, auch HOMOD genannt. Der Hold & Modify-Modus kann nur mit gering auflösenden Bildern im Single-Playfield-Betrieb verwendet werden. Ansonsten wäre die Playfield-Hardware aus Geschwindigkeitsgründen überfordert. Die Prozedur zum Setzen von Display-Window und Datafetch entsprechen denen, die auch bei »normalen« Playfields verwendet werden.

Folgende Bits in BPLCON0 müssen für einen ordnungsgemäßen Betrieb des HAM-Modus berücksichtigt werden:

- Bit 11, HOMOD, hat den Wert 1.
- Bit 10, DBLPF, ist 0 (Single Playfield)
- Bit 15, HIRES, ist 0 (Low Resolution)
- Bits 14 bis 12, BPU2 bis BPU0, haben entweder den Wert 110 oder 101 (5 oder 6 Bitplanes).

Das Setzen der Bitplanes in Hold & Modify-Modus ist relativ schwierig, da man beim Zeichnen eines Punktes immer den vorherigen berücksichtigen muß. Damit ist auch keine Animation möglich, weil die Playfield-Hardware dazu zu langsam ist. Der HAM-Modus eignet sich damit am besten für Standbilder, die besonders farbintensiv sind; zum Beispiel eindrucksvolle Titelbilder für Softwarepakete oder Grafiken, die besonders viele Schattierungen einer Farbe benötigen. Animation mit HAM-Bildern zu produzieren, wird nicht empfohlen (Vertikales Scrolling ist möglich; aus Zeitgründen geht dann aber nichts anderes mehr, da die Playfield-Hardware

wegen des ständigen DMA-Zugriffs auf das RAM den Speicherzugriff des Prozessors blockiert).

6.6 Verschiedenes zum Thema Playfields

6.6.1 Screens - unterschiedliche Playfields gleichzeitig

Die Fähigkeit des Amiga, verschiedene Playfields auf virtuellen Bildschirmen (den sogenannten Screens) darzustellen und einfach mit der Maus auf- und abzuziehen, ist ein Produkt der Grafikbibliotheken des Amiga-Betriebssystems. Die Playfield-Hardware tut dabei nichts anderes, als das Datafetching bis zu dem Punkt zu führen, an dem der nächste Screen beginnt. An dieser Stelle wird der Copper von der Systemsoftware verwendet, um alle Register auf die Werte des anderen Playfields zu setzen. Hier wird das Data-Fetching fortgeführt, jedoch mit den Werten des anderen Playfields. Die virtuellen Bildschirme sind also reine Softwaresache, die jedoch die Playfield-Hardware und die Fähigkeiten des Coppers, Register zu ändern, ausnutzt. Wie so eine mehrteilige Grafik aussehen kann, zeigt das folgende Bild.

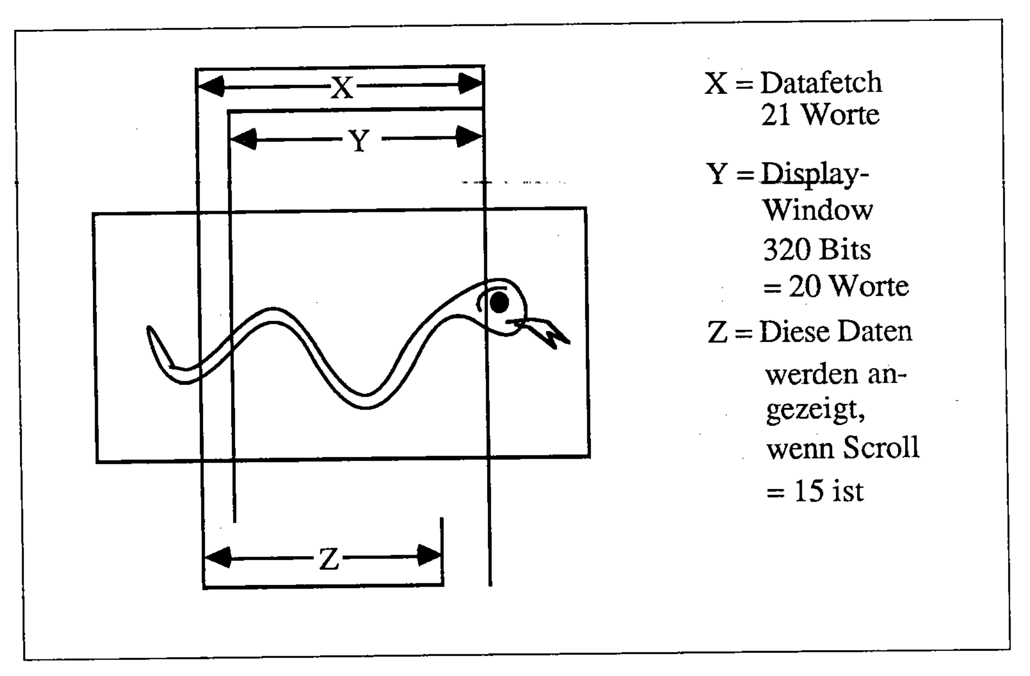


Bild 6.10: Virtuelle Bildschirme

6.6.2 Interaktion zwischen Playfields und anderen Objekten

Playfields können das Display mit Sprites teilen. Die sogenannten BOBs werden von der Hardware als Teile der Playfields betrachtet und sind eine reine Softwareangelegenheit. Sprites dagegen werden unabhängig von den Playfields dargestellt. Sprites und Playfields haben jedoch Prioritäten untereinander. (Siehe dazu auch den Abschnitt »Kontrollhardware«.) Sprites können Playfields überlappen, aber auch von ihnen überdeckt werden. Den Playfields kann dabei eine Priorität gegeben werden, die relativ zu der der Sprites angegeben wird. Kollisionen zwischen Playfields und Sprites werden ebenfalls von der Kontrollhardware registriert.

6.6.3 Benutzung einer externen Videoquelle

Die Hardware des Amiga ist darauf vorbereitet, ein externes Videosignal zu verwenden und anstelle der Hintergrundfarbe (COLOR0) anzuzeigen. Dazu gibt es eine Erweiterung, das sogenannte Genlock-Interface. »Genlocking« ist nichts anderes, als eine zweite Videoquelle mit einer Videoquelle zu synchronisieren. Das Genlock-Interface synchronisiert also ein Bild eines Laserdiskspielers, eines Videorekorders oder einer Kamera mit dem, das der Amiga ausgibt. Damit der Amiga das externe Signal anstelle der Farbe 0 anzeigt, muß Bit 1 des BPLCON0-Registers auf den Wert 1 gesetzt werden. Dieses Bit hat den Namen ERSY, was für »ExteRnal SYnchronised Signal« steht.

7

Die Sprite-Hardware

Sprites sind Grafikobjekte, die völlig unabhängig vom Playfield-Display und unabhängig voneinander angezeigt werden. Das heißt, egal was wir auf dem Bildschirm haben (LoRes, HiRes, Scrolling, etc.), die Sprites werden unabhängig davon auf den Bildschirm gebracht. Die Sprites werden durch 8 speziell dafür vorgesehene DMA-Kanäle in die Videologik »eingespeist«, um so die Playfields zu überlagern oder unter sie gebracht zu werden.

Ein einfaches Sprite ist 16 Pixels breit und beliebig lang, und darf höchstens drei Farben plus Transparenz (also vier Farbkombinationen) benutzen. Für größere Objekte beziehungsweise Sprites mit mehreren Farben, kann man Sprites verknüpfen. Da man die DMA-Kanäle mehrmals pro Rasterstrahldurchlauf verwenden kann, ist es auch möglich, mehr als 8 Sprites gleichzeitig darzustellen.

7.1 Sprites positionieren

Eine Spriteposition auf dem Bildschirm wird definiert durch seine (x,y)-Koordinaten. Position (0,0) ist die linke obere Ecke des Displays. Durch den sogenannten »Overscan« ist die Koordinate (0,0) im Normalfall aber nicht sichtbar. Die Sichtbarkeit des Sprites ist außerdem abhängig von der Größe des Display-Windows. Die Spriteposition wird immer angegeben als LoRes-NonInterlace-Koordinate, unabhängig vom Grafikmodus des Playfield-Hintergrundes. Die Spritekoordinaten repräsentieren dabei den Beginn der linken oberen Ecke der Spritedaten.

Die horizontale Position eines Sprites (die x-Koordinate) kann an jedem Punkt des Bildschirms zwischen 0 und 447 liegen. Um sichtbar zu sein, muß ein Sprite allerdings innerhalb der Grenzen des Display-Windows liegen. Der normalerweise nutzbare Bereich des Videoschirms ist von Pixel 64 bis Pixel 383 (320 benutzbare Pixels). Wird ein x-Wert außerhalb dieses Bereichs angegeben, wird das Sprite dort »abgeschnitten«.

Um mit den Koordinaten eines LoRes-Bildes übereinzustimmen, muß der Wert 64 zum x-Wert addiert werden. Wollen wir zum Beispiel ein Sprite 80 Pixels rechts vom Bildanfang positionieren, errechnen wir die benötigte x-Koordinate durch:

Gewünschte x-Position 80 + nicht sichtbarer Bereich 64 = Position 144.

Die Koordinate wird an den Anfang der Sprite-Datenstruktur geschrieben, die das Sprite beschreibt. (Zur dieser Datenstruktur später noch mehr.) Beachten Sie, daß die x-Position nicht etwa die Mitte, sondern das am weitesten links liegende Pixel des Sprites beschreibt. Die linke Position bedeutet hier das am weitesten links liegende Bit der 16 Bit breiten Datenstruktur eines Sprites, und nicht des am weitesten links sichtbaren Pixels (die links liegenden Pixels können ja auch transparent sein).

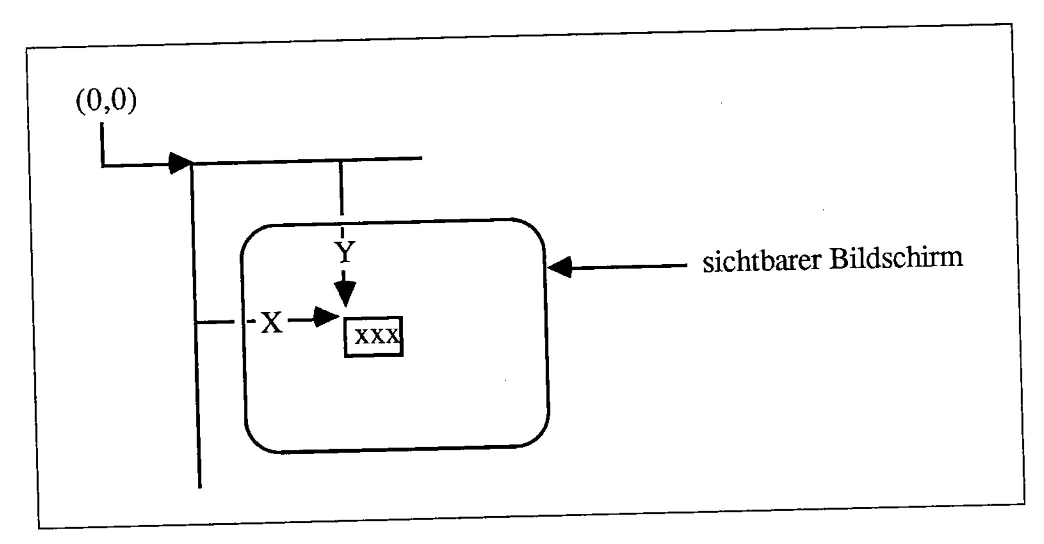


Bild 7.1: Die Spriteposition

Für die y-Koordinate ist jeder Wert zwischen 0 und 262 möglich, der normalerweise benutzbare Bereich liegt zwischen 44 und 243 (beim PAL-Amiga zwischen 44 und 256). Wird eine y-Koordinate von weniger als 44 angegeben, erscheint nur der Teil des Sprites, der unter Position 44 liegt. Soll ein Sprite also 33 Pixels unterhalb des Bildanfangs liegen, berechnen wir die y-Position mit:

Gewünschte y-Position 33 + nicht sichtbarer Bereich 44 = y-Koordinate 77.

Dieser Wert muß ebenfalls in die Sprite-Datenstruktur geschrieben werden. Sowohl die x- als auch die y-Koordinate sind je 9 Bits lang und – wie später noch näher beschrieben – auf mehrere Datenwörter verteilt.

7.2 Wie ein Sprite aufgebaut ist

Ein Sprite kann jede Form haben, die in eine 16-Pixel-Breite paßt. Das folgende Bild zeigt, wie so ein Sprite zum Beispiel aussehen könnte.

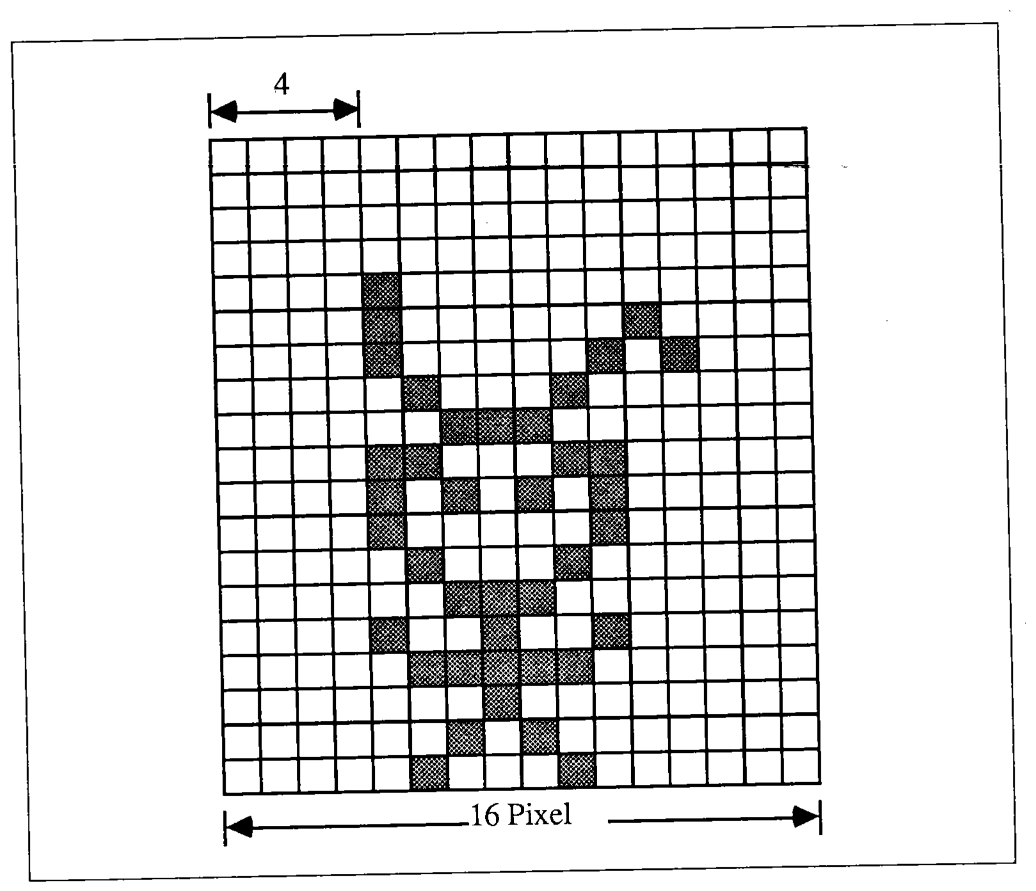


Bild 7.2: Ein einfaches Sprite

Die Form dieses Sprites könnten wir natürlich auch um bis zu vier Bits nach links oder bis zu drei Bits nach rechts innerhalb der 16-Bit-Matrix verschieben. Das Sprite sieht dann immer noch gleich aus, nur die Position des Objektes verschiebt sich um entsprechend viele Pixels relativ zur angegebenen x-Position, die ja nur den Anfang der 16-Bit-Matrix und nicht den der Form, die wir darstellen wollen, angibt.

7.2.1 Farben von Sprites

Die Farben innerhalb von Sprites stellen wir durch Überlagern einzelner Datenworte her. Im Normalfall, also wenn die Sprites

nicht durch »Attach« verbunden sind (dazu später mehr), ergeben immer zwei Worte eine Spritezeile. Die Bitkombinationen der Wortpaare ergeben eine Zahl von 0 bis 4.

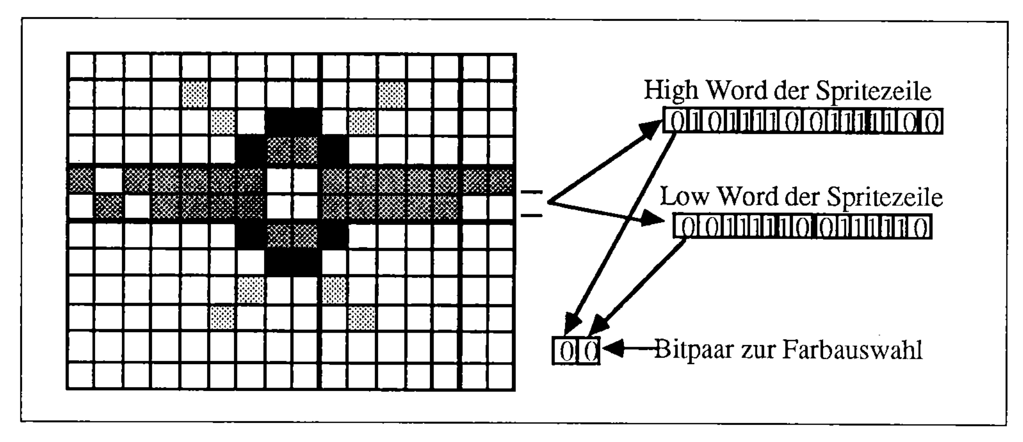


Bild 7.3: Die Speicherorganisation eines Sprites

Die Kombination zeigt auf eines der vier Farbregister, die dem jeweiligen Sprite-DMA-Kanal zugeteilt sind. Die 8 Sprites benutzen Farbregister 16 bis 31. Für Zwecke der Farbauswahl sind die Sprites in Gruppen von je zwei Stück aufgeteilt. Jedes Paar benutzt vier Farbregister wie in der in Abbildung 7.4 angegebenen Reihenfolge.

Beachten Sie bitte, daß die Wahl des ersten Registers einer jeden 4-Register-Gruppe (also Bitkombination 00) nicht die darin enthaltene Farbe wiedergibt, sondern Transparenz anzeigt.

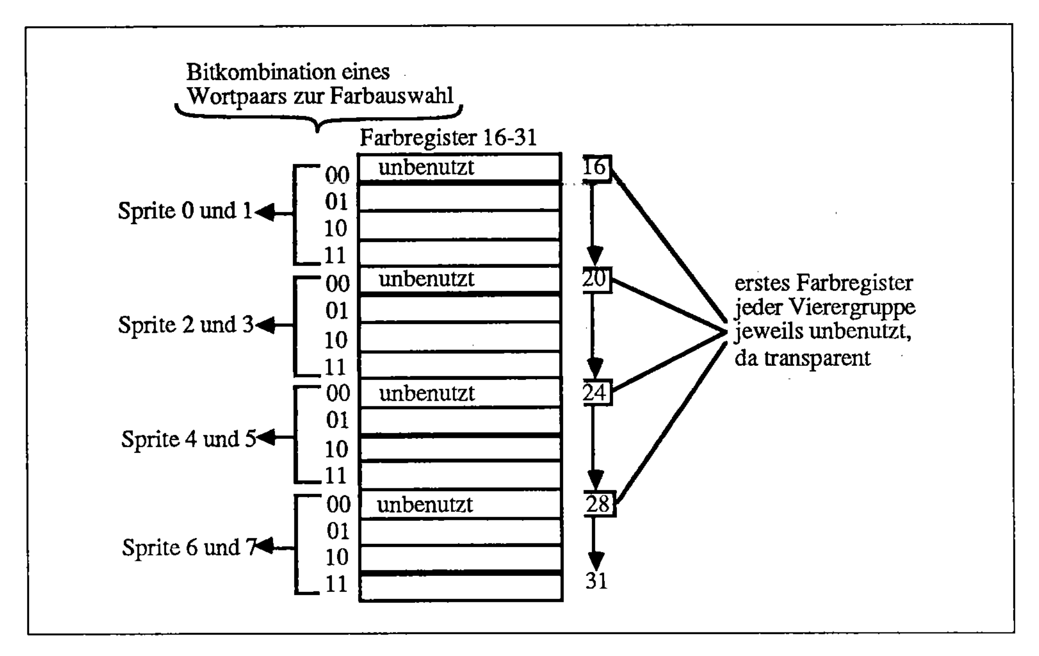


Bild 7.4: Die Zuordnung von Farben zu Spritegruppen

Der Transparenz-Modus zeigt diejenige Farbe an, die das Objekt mit der nächstniedrigen Videopriorität hat (als das »darunter« liegende Sprite oder Playfield). Sprites haben feste, durch die Spritehardware bedingte Prioritäten untereinander. Die Prioritäten zwischen Sprites und Playfields sind wahlweise einstellbar.

7.2.2 Die Datenstruktur eines Sprites

Ist einmal das Aussehen eines Sprites definiert, wird daraus eine Datenstruktur gemacht, mit der die Spritelogik etwas anfangen kann. Die Struktur von Sprites besteht aus einer Reihe von 16-Bit-Worten, die hintereinander im Speicher liegen. Einige der Worte beinhalten Kontrollinformation zum Sprite, zum Beispiel Koordinaten. Folgende Schritte sind notwendig, um eine Spritedatenstruktur ordnungsgemäß herzustellen:

- Schreiben Sie die horizontale und vertikale Position des Sprites in das erste Kontrollwort.

- Schreiben Sie die vertikale Endposition in das zweite Kontrollwort.
- Erstellen Sie die Datenworte durch Umsetzung des Grafikentwurfs in binäre Werte, wobei jeweils eine Spritezeile aus zwei Datenwörtern zusammengesetzt ist.
- Schreiben Sie die Kontrollworte, die das Ende der Spritedatenstruktur anzeigen.

Die Spritedaten müssen – wie alle Daten für die Custom Chips – in den unteren 512K RAM liegen. Halten Sie genügend Speicherplatz frei für Ihre Sprites! Der notwendige Speicherplatz errechnet sich folgendermaßen:

2 Worte Kontrollinformation
+ Anzahl der Spritezeilen mal zwei (2 Worte pro Zeile)
+ 2 Worte »End-of-Data«-Kennzeichen

= Notwendige Anzahl Speicherworte für ein Sprite

Die entsprechenden Kontroll- und Datenworte haben von den Entwicklern des Amiga Namen bekommen, die im großen und ganzen denen der Register ähneln. Diese Worte heißen:

SPRxPOS – Sprite Control Word 1

Dieses Wort beinhaltet die vertikale (VSTART) und horizontale (HSTART) Startposition des Sprites. An dieser Stelle wird die oberste Zeile des Sprites angezeigt.

Bits 15-8 enthalten die niederen 8 Bits von VSTART (neuntes Bit im zweiten Kontrollwort).

Bits 7-0 enthalten die 8 höherwertigen Bits von HSTART (niederstwertiges Bit in Kontrollwort 2).

SPRxCTL – Sprite Control Word 2

Dieses Wort gibt die Stop-Position des Sprites an (das heißt die Länge des Sprites). Außerdem beinhaltet dieses Kontrollwort weitere Informationen zur Startposition sowie über »Sprite Attachment« (siehe dazu auch den nachfolgenden Abschnitt).

Bits 15-8 enthalten die niederwertigen 8 Bits von VSTOP.

Bit 7 enthält das Attach-Bit zur Kombination zweier Sprites.

Bits 6-3 sind nicht benutzt (aufgrund von Aufwärtskompatibilität sollte man diese Bits auf Null setzen).

Bit 2 ist das höchstwertige VSTART-Bit.

Bit 1 ist das höchstwertige VSTOP-Bit.

Bit 0 ist das niederstwertige Bit der HSTART-Information.

Die Anzahl der Spritezeilen wird durch VSTOP – VSTART + 1 definiert.

Spritedatenworte

Für jede Zeile des Sprites benötigen wir zwei Datenworte. Diese zwei Datenworte teilen wir auf in Low Word und High Word. Obwohl im Speicher zuerst das Low Word folgt, enthält bei der Kombination aus Low- und High Word das High Word das erste Bit der binären Farbauswahlnummer. Das Low Word beinhaltet das zweite Bit der Farbwahlnummer. Wie so ein Farbsprite aufgebaut ist, haben wir bereits besprochen. Sprite-DMA-Kanal 0 benutzt Farbregister 16 bis 19, Spritekanal 1 greift auf Farbregister 20 bis 23 zu und so weiter.

Datenendanzeiger

Die Datenendanzeiger oder auch »End-of-Data-Words« zeigen das Ende der Spritedateninformation an. Wenn die vertikale Position des Rasterstrahls gleich VSTOP ist, werden die nächsten zwei Worte aus der Spritedatenstruktur in die Spritekontrollregister geladen, anstatt an die Farblogik gesendet zu werden. Diese zwei Worte werden von der Hardware so interpretiert wie auch die beiden ersten Kontrollworte der Datenstruktur. Ist der VSTART-Wert dieser Kontrollworte geringer als der vorher in die Spriteregister eingelesene Wert, wird dasselbe Sprite nicht noch einmal angezeigt. Soll ein Sprite-DMA-Kanal das Sprite nur einmal anzeigen, sollte man die beiden letzten Worte mit Nullbits füllen. (Die Mehrfachanzeige von Sprites wird später noch näher erläutert.)

Kluge Köpfe werden schon jetzt sehen, daß Sprites auf ähnliche Weise auf den Bildschirm gebracht werden wie Playfields. Das Datafetching funktioniert aber etwas anders, da hier erst Kontrollworte in Register gebracht werden und anschließend der eigentliche Display-Vorgang beginnt. Ist das Sprite zu Ende angezeigt, werden

zwei weitere Kontrollworte in die Register der Spritelogik eingelesen.

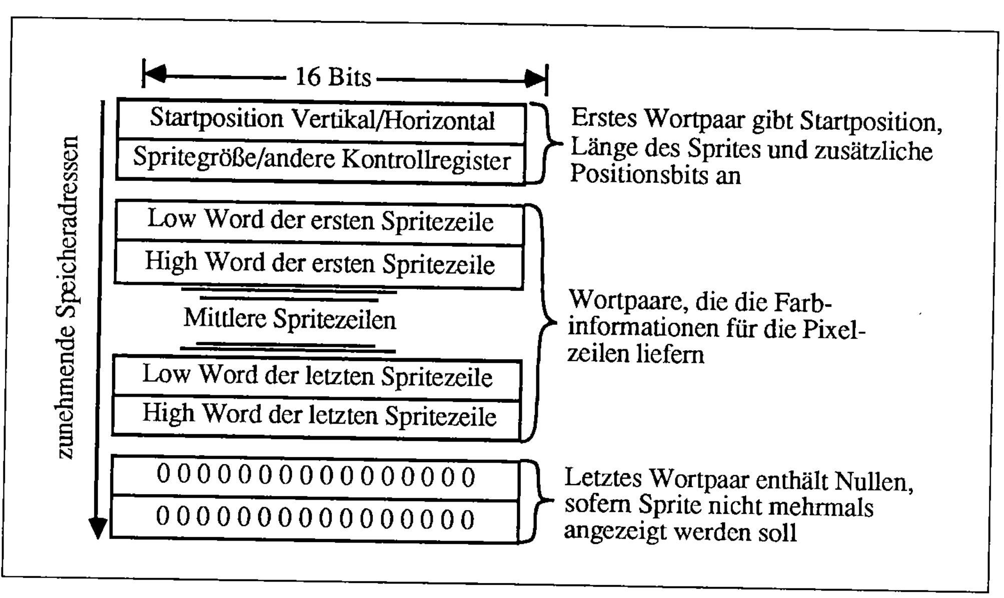


Bild 7.5: Die Spritedatenstruktur

7.3 Das Sprite kommt auf den Bildschirm

Nachdem die Spritedatenstruktur definiert und im Speicher abgelegt ist, muß der Hardware natürlich irgendwie mitgeteilt werden, wie diese Daten angezeigt werden sollen. Die Anzeige von Sprites kann auf zwei grundsätzlich verschiedene Methoden geschehen: Im automatischen und im manuellen Modus. Uns interessiert hier nur die automatische Methode. Hier geschieht die Anzeige der Sprites per DMA: Der Sprite DMA-Kanal holt die Daten aus dem Speicher und zeigt sie an, solange bis die VSTOP-Position erreicht ist. Die Schritte, ein Sprite darzustellen, sind folgende:

Zuerst muß der gewünschte DMA-Kanal angewählt werden (Sprite DMA 0 bis 7). Die Spritezeiger (Adreßregister) müssen wir so einstellen, daß die Hardware weiß, wo im Speicher die Spritedatenstruktur liegt. Nun schalten wir die Sprite-DMA an. Das

Sprite wird dann angezeigt, aber nur einen Bildaufbau lang. Um es ständig sichtbar zu haben, müssen während des vertikalen Blankings die Spritezeiger immer wieder auf den Startwert zurückgesetzt werden.

7.3.1 Die Adreßzeiger und das Spritedaten-Fetching

Bei der Entscheidung, welcher DMA-Kanal gewählt wird, sollten die Videoprioritäten berücksichtigt werden. Die Spritedaten und Kontrollworte, die nun bereits im Speicher stehen, muß sich der DMA-Kanal aus dem Speicher holen und bearbeiten. Dazu benutzt er zwei Zeiger. Während des vertikalen Blanking vor dem ersten Anzeigen des Sprites muß die Speicheradresse der Spritedatenstruktur in diese Zeiger geschrieben werden. Für jedes Sprite gibt es zwei solche Zeiger, genannt SPRxPTH und SPRxPTL.

SPRxPTH enthält die höherwertigen drei Bits der Speicheradresse, an der das erste Wort des Sprites liegt. SPRxPTL enthält die niederwertigen 15 Bits. Wie üblich darf auch hier einfach nur ein Longword in SPRxPTH geschrieben werden.

Diese Zeiger sind – wie auch die Bitplanezeiger der Playfieldhardware – dynamisch. Sie werden durch den Sprite-DMA-Kanal inkrementiert, um auf die Kontrollworte zu zeigen, sie in andere Register zu speichern, dann auf die Datenworte zu zeigen und schließlich auf die Datenende-Worte. Nachdem die Kontrollinformation eingelesen ist und in anderen Registern gespeichert ist, wird inkrementiert. Nach dem Inkrement zeigen die Spritepointer auf die Datenworte. Diese werden in Spritedatenregister eingelesen, die der DMA-Kanal dazu benutzt, sie auf dem Bildschirm anzuzeigen. Auch das ist eine Art »Datafetching«, wie sie bereits im Abschnitt »Playfield-Hardware« beschrieben wurde.

Auf diese Weise wird das Sprite nur einmal kurz angezeigt. Soll das Sprite länger auf dem Schirm bleiben, müssen nach jedem Rasterstrahldurchlauf die Adreßzeiger wieder zurückgesetzt werden. Das Zurücksetzen der Spritezeiger kann als Teil der Vertical Blanking Routine behandelt werden (siehe auch »Copperlisten«).

7.3.2 Die Bewegung eines Sprites

Die Bewegung eines Sprites erzeugen wir einfach durch die Änderung der Werte VSTART und HSTART innerhalb der Spritedatenstruktur. Nach der Änderung des Wertes werden die Daten durch den DMA-Kanal wiedergelesen und das Sprite automatisch wieder angezeigt, diesmal jedoch an der neuen Position.

Spricht man die Hardware direkt an und nicht über die Betriebssystemroutinen, muß man darauf achten, daß man die Spriteposition nicht zur selben Zeit ändert, zu der die Hardware auf das Kontrollwort zugreift. Während die Daten von der Hardware eingelesen werden, kann es nämlich in ungünstigen Fällen dazu kommen, daß die alte Startposition eingelesen wird, dann die Kontrollworte geändert werden und schließlich die Stop-Position der neuen Position eingelesen und verwendet wird. Dadurch erscheint »Datenmüll« auf dem Bildschirm. Deshalb sollte man die Kontrollworte nur dann ändern, wenn man weiß, daß das System nicht zur selben Zeit darauf zugreift. Die sicherste Zeit dazu ist die des vertikalen Blankings. Damit wird auch die Spritebewegung ein Teil des vertikalen Blanking-Interrupts.

7.3.3 Mehrere Sprites

Um mehr als ein Sprite darzustellen, entwickeln wir eine Datenstruktur für jedes und behandeln das Anzeigen der Sprites für jeden DMA-Kanal so wie auch bei einem einzigen Sprite. Die Zeiger für Sprite 1 sind dann also SPR1PTH und SPR1PTL, für Sprite 2 SPR2PTH und SPR2PTL und so weiter. Insgesamt erhalten wir so 8 Sprites, die wir auf diese Weise auf den Bildschirm bringen können.

Bitte beachten Sie, daß beim Anschalten des Sprite DMA alle Spritekanäle aktiviert werden und im »Automatic Mode« sind, also von selbst angezeigt werden. Wenn Sprite-DMA einmal eingeschaltet ist, müssen alle 8 Spritezeiger auf entweder ein echtes Sprite oder auf sichere »Nullsprites« gesetzt werden. Nicht initialisierte Spritezeiger können recht seltsame Effekte auf dem Bildschirm erzeugen.

Beim Erstellen mehrerer Sprites muß darauf geachtet werden, welches Sprite »vor« und welches Sprite »hinter« anderen Sprites

erscheint. Die Prioritäten der Sprites untereinander sind durch die Hardware festgelegt: Sprite 0 erscheint ganz vorne, Sprite 1 dahinter, bis zu Sprite 7, das von allen anderen überlagert wird (sofern die Positionen gleich sind). Siehe dazu auch das folgende Bild.

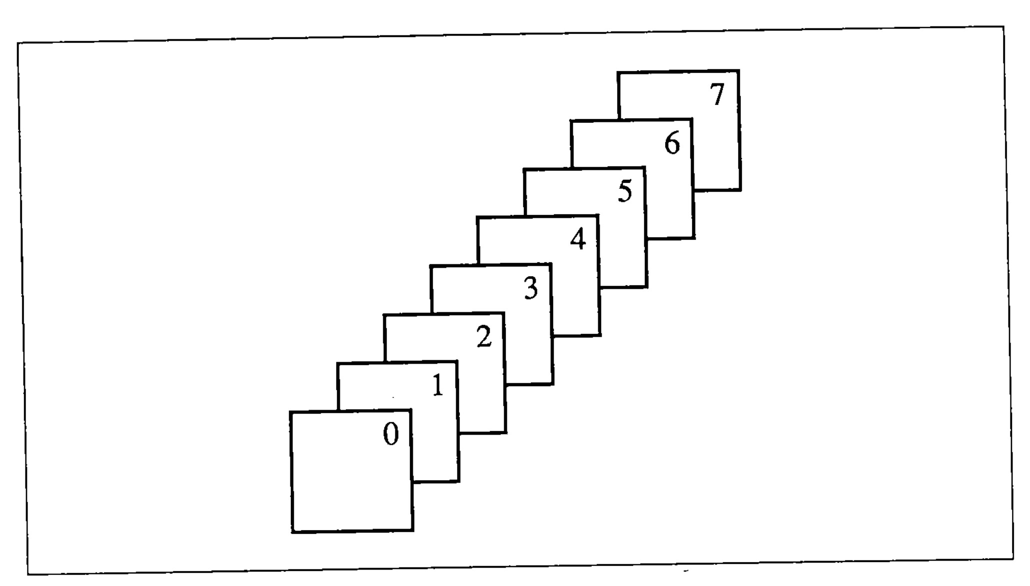


Bild 7.6: Spriteprioritäten

7.4 VSprites – oder wie man mehr als 8 Sprites ins Bild bringt

Jeder DMA-Kanal kann während eines Rasterstrahldurchlaufs mehrmals benutzt werden. Das bedeutet, wir können mit einem Spritekanal mehrere Sprites anzeigen, sofern sie auf dem Bildschirm untereinander liegen. Das folgende Bild zeigt, wie so etwas aussehen kann.

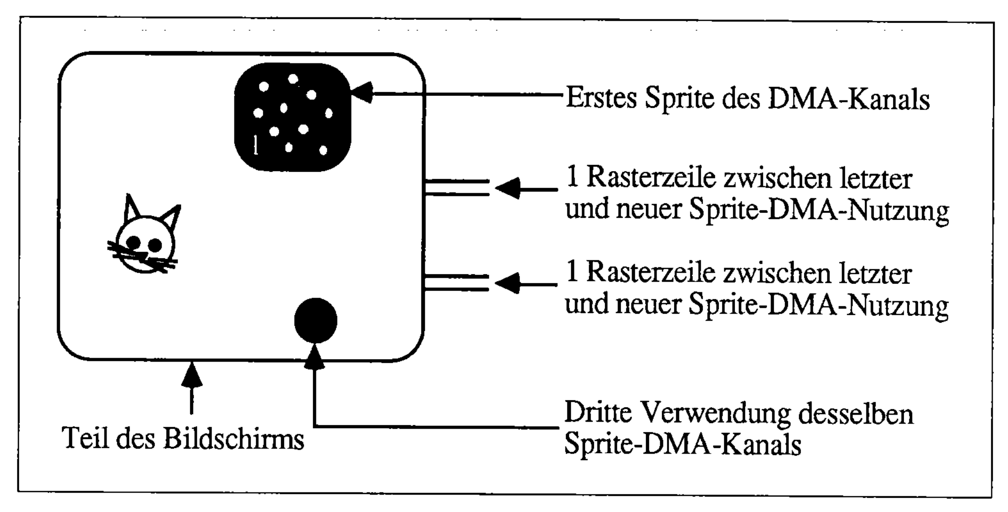


Bild 7.7: Anzeigen mehrerer Sprites durch denselben DMA-Kanal

Wie bekannt, werden am Ende eines Sprites je zwei Nullworte an das Ende der Datenstruktur gehängt, damit der DMA-Kanal keine weiteren Daten während des Rasterstrahldurchlaufs anzeigt. Um weitere Daten anzeigen zu können und somit mehr Sprites auf den Bildschirm zu bringen, setzen wir anstatt der zwei Nullworte eine weitere Spritedatenstruktur an das Ende der ersten. Das können wir beliebig oft machen, bis der Rasterstrahl am unteren Ende des Bildschirms ist. Die zwei Nullworte hängen wir dann an das Ende der großen »Mehrfachspritestruktur«. Die einzige Beschränkung dabei ist, daß wir auf diese Weise nicht zwei Sprites desselben DMA-Kanals auf selber Höhe darstellen können. Die unterste Linie eines Sprites und die oberste Linie des nachfolgenden Sprites müssen mindestens eine Rasterzeile zwischen sich haben. Diese Beschränkung ist notwendig, weil nur zwei DMA-Zyklen pro Linie für jeden Spritekanal verfügbar sind. Der Spritekanal benötigt die Zeit während der leeren Zeile, um das Kontrollwort des nachfolgenden Sprites zu lesen. Und außerdem können wir den Rasterstrahl nicht wieder nach oben lenken. Das folgende Bild zeigt den Aufbau einer großen Datenstruktur, wie sie bei der Mehrfachnutzung von DMA-Kanälen zu entwickeln ist.

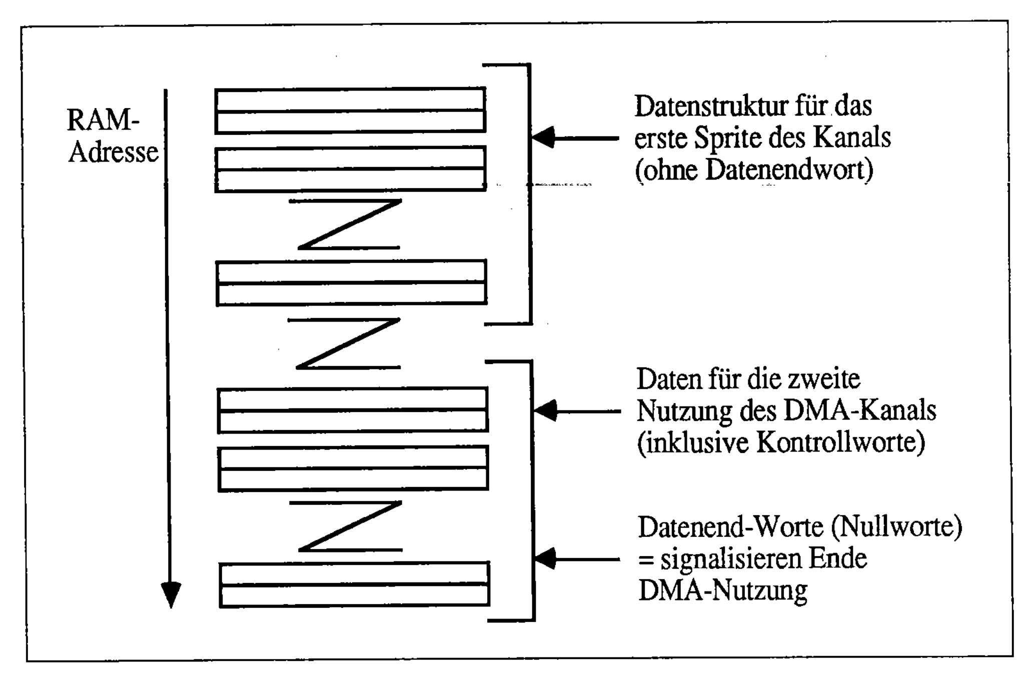


Bild 7.8: Datenstruktur für Mehrfach-DMA-Nutzung

Mit Hilfe dieser Technik läßt sich zum Beispiel ohne Probleme ein Space-Invaders-Spiel programmieren: eine Datenstruktur beschreibt 12 untereinander liegende verschiedene Invaders. Da wir in jeder Zeile für unser Spiel 8 Invaders verwenden, haben wir insgesamt 12 * 8 = 96 Sprites. Zu definieren brauchen wir jedoch nur eine einzige Datenstruktur für 12 untereinanderliegende Sprites; die Zeiger der anderen 7 DMA-Kanäle zeigen auf dieselbe Struktur, so daß wir praktisch 96 Fliegen (Sprites) mit einer Klappe (Datenstruktur) schlagen. Anhand dieses Beispiels sieht man, daß die Spritehardware des Amiga ein geniales Stück Technik ist.

7.5 Sprite-Attach – mehr Farben durch Kombination von Sprites

Im Normalfall hat ein Sprite 3 Farben plus Transparenz. Die Farbpalette kann jedoch auf 15 Farben erweitert werden, indem zwei Sprites zusammengefaßt werden. Diese Zusammenfassung nennt man

»Attached Sprites«, wörtlich übersetzt also »verbundene Sprites«. Um die größere Farbwahl zu ermöglichen, sind folgende Schritte notwendig: Für ein Attach-Sprite müssen zwei DMA-Kanäle verwendet werden, wobei wir zwei Sprites derselben Größe entwerfen, die zugleich auf derselben Position liegen müssen. Dazu muß das Bit ATTACH im zweiten Spritekontrollwort gesetzt werden, und voila, schon haben wir doppelt so viel Farben – und halb so viele Sprites!

Die 15 Farben (plus Transparenz) werden aus den Farbregistern 17 bis 31 ausgewählt. Da durch das Zusammenfassen zweier Sprites vier Bits für die Farbauswahl verfügbar sind, haben wir 16 Farbmöglichkeiten. Zur Berechnung des richtigen Farbregisters nehmen wir einfach den binären Wert der Spritebitkombinationen in dezimal plus 16. Farbregister 16 wird allerdings nicht benutzt, sondern gibt (Kombination 0000) Transparenz. Die letzte Kombination der Reihe, 1111, ergibt somit 15 + 16 = 31.

Es können jeweils Sprite 0 und 1, 2 und 3, 4 und 5, 6 und 7 zusammengefaßt werden. Andere Kombinationen sind nicht möglich. Die in der Viererkombination links stehenden beiden Bits werden immer durch das Sprite mit der höheren Nummer gestellt, die beiden niederwertigen Bits durch das Sprite mit der niedrigeren Nummer. Haben wir beispielsweise Sprite 1 zu Sprite 0 »attached«, also dazukombiniert, wird eine Spritezeile in folgender Reihenfolge gebildet durch:

1. High Word der ersten Datenzeile von Sprite 1
2. Low Word der ersten Datenzeile von Sprite 1
3. High Word der ersten Datenzeile von Sprite 0
4. Low Word der ersten Datenzeile von Sprite 0

Die Zusammenfassung ist nur aktiv, wenn das ATTACH-Bit (Bit 7 des zweiten Kontrollwortes) des Sprites mit der ungeraden Kanalnummer auf 1 gesetzt ist. In unserem Beispiel muß also Bit 7 des zweiten Kontrollwortes von Sprite 1 gesetzt werden. Ein Setzen dieses Bits in Sprite 0 ist nicht notwendig und bewirkt auch nichts.

Wenn die Sprites bewegt werden, müssen beide immer an derselben Position gehalten werden. Im Normalfall geschieht dies durch die Copperlisten. Sind die Positionen nicht gleich, ändern ihre Pixels die Farben zur üblichen Farbmenge von 3 Farben plus Transparenz. Die Farbwahl verläuft dabei allerdings anders, als bei nicht zusammengefaßten Sprites: Das Sprite mit der niedrigeren Nummer

benutzt die Farben der Register 17 bis 19, das Sprite mit der höheren Nummer verwendet Register 20, 24 und 28. Damit können zusammengefaßte Sprites auch einzeln bewegt werden; das ist zwar nicht sehr sinnvoll, kann aber schöne Farbeffekte erzeugen, wenn zum Beispiel die Sprites abwechselnd übereinander liegend und voneinander versetzt dargestellt werden.

7.6 Zusammenfassung aller Spriteregister

Insgesamt gibt es acht ganze »Sets« von Registern. Jedes Set beinhaltet 5 Register. Für jeden DMA-Kanal haben wir damit ein 5–Register-Set. Für jedes Set ist die hier genannte Variable »x« mit einer Zahl von 0 bis 7 (je nach DMA-Kanal) zu ersetzen.

Die Spriteregister sind in drei grundsätzliche Arten aufgeteilt: Datenregister, Adreßzeiger und Kontrollregister.

7.6.1 Adreßzeiger

Zeiger oder »Pointers« sind Register, die durch das System verwendet werden, um auf die gegenwärtig benutzten Daten zu zeigen. Während der Bildanzeige werden diese Register inkrementiert, um auf die benötigten Daten zu zeigen, während der Bildaufbau fortschreitet. Während des vertikalen Blankings müssen diese Register neu geschrieben (zurückgesetzt) werden, damit beim nächsten Bildaufbau wieder ein Sprite erscheint. Ein Wechseln zwischen zwei Sprites durch abwechselndes Setzen der Sprite-Adreßzeiger auf verschiedene Datenstrukturen kann einen Animationseffekt hervorrufen (zum Beispiel Flattern eines Vogels).

SPRxPTH und SPRxPTL

Dieses Registerpaar enthält eine 18-Bit-Adresse der Daten von Sprite x. Die Register beinhalten die 3 höherwertigen Bits (PTH) und 15 niederwertigen Bits (PTL) der Adresse. 68000-Programmierer können auch ein Longword in SPRxPTH schreiben.

7.6.2 Kontrollregister

SPRxPOS

Positionsregister für Sprite x. Das Wort, das in dieses Register geschrieben wird, gibt die linke obere Ecke des Sprites, sprich die Position des dem höchstwertigen Bit des ersten Datenwortes entsprechenden Pixels, an. Dabei muß beachtet werden, daß die Daten für LoRes-NonInterlace-Auflösung gelten, da die Spriteauflösung unabhängig von der Bitplane-Auflösng ist.

Bits 15-8 enthalten die vertikale Startposition, Bits V7 bis V0.

Bits 7-0 enthalten die horizontale Startposition, Bits H8 bis H1.

Die Positionen sind jeweils 9-Bit-Werte (neuntes Bit siehe SPRxCTL).

Dieses Register wird normalerweise vom DMA-Kanal selbst beschrieben. Es beinhaltet Kontrollinformation für den Spritedaten-Fetch-Prozeß.

SPRxCTL

Auch dieses Register wird im Normalfall nur durch den DMA-Kanal beschrieben. SPRxCTL beinhaltet ebenfalls Kontrollinformation für das Datafetching.

Bits 15-8 enthalten die vertikale Stop-Position, Bits V7 bis V0.

Bit 7 ist das ATTACH Bit.

Dieses Bit ist nur wirksam für ungerade DMA-Kanal-Nummern, da nur Sprites ungerader Zahl an Sprites gerader Zahl gekoppelt werden können. Durch das Koppeln ist eine größere Farbwahl möglich; während des Attach-Modus werden die Sprites normalerweise gleichzeitig unter Prozessorkontrolle bewegt. Die Sprites, obwohl zusammengefaßt, können jedoch unabhängig voneinander bewegt werden. Die größere Farbwahl ist jedoch nur möglich, wenn die Positionen identisch sind.

Bits 6-3 sind reserviert für zukünftige Erweiterungen (auf Null setzen).

Bit 2 ist das Bit V8 des VSTART-Wertes.

Bit 1 ist das Bit V8 des VSTOP-Wertes.

Bit 0 ist das Bit H0 des HSTART-Wertes.

7.6.3 Datenregister

SPRxDATA und SPRxDATB

Diese Spritedatenregister werden im Normalfall nur durch Displaylogik benutzt und per DMA eingelesen und ausgegeben. Sie enthalten die gerade auszugebenden Daten des Displays.

8 Der Blitter

Der Blitter ist derjenige Teil der Hardware, der die Grafik des Amiga so schnell macht. Der Blitter ist quasi der »Datenschaufler« im Amiga. Die Operationen, die er durchführt, sind wesentlich schneller als es entsprechende mit dem 68000 geschriebene Routinen wären. Da das Setzen der Register des Blitters aber auch immer Zeit (Prozessorzeit!) benötigt, sollte man bei vielen kleinen Datenmengen überlegen, ob man nun den Blitter einsetzt oder nicht. Die Systemsoftware nutzt ihn dort, wo er wirklich Geschwindigkeitsvorteile bringt: Bei der Animation von größeren Objekten, den BOBs, bei der Fensterverwaltung (zum Beispiel beim Puffern im Smart-Refresh-Modus) und auch beim Ziehen längerer Linien und Füllen von Flächen.

Beim Kopieren von Daten kann der Blitter diese Daten auch logisch miteinander verknüpfen und bietet damit eine Vielzahl von Möglichkeiten in der Grafik. Die ersten Teile dieses Abschnitts beschreiben ausführlich die Kopierfunktionen des Blitters. Das erscheint auf den ersten Blick vielleicht nicht als eine Grafikfähigkeit – doch das schnelle Kopieren von Daten ist die Grundlage für die sogenannten BOBs und deren schnelle Animation. Die letzten Teile des Abschnitts beziehen sich dann hauptsächlich auf das Linienziehen.

Hier eine kurze Zusammenfassung dessen, was man mit dem Blitter alles tun kann:

- Daten kopieren: Der Blitter kann Bitplane-Inhalte überall in die unteren 512KByte des Speichers kopieren.

- Mehrere Datenquellen kombinieren: Anstatt nur Daten aus einer einzigen Quelle zu holen, kann der Blitter bis zu drei Datenquellen nehmen und diese zusammengefaßt durch logische Operationen (siehe unten) in einen Zielspeicher kopieren. Das erlaubt zum Beispiel eine Überlappung von BOBs (Softwaresprites) in gewünschter Reihenfolge ohne Berechnungen durch den 68000.

- Logische Operationen durchführen: Der Blitter kann während des Kopiervorganges bis zu 256 logische Operationen an den drei Datenquellen vornehmen.

- Animation durch »Shifting«: Der Blitter kann eine oder zwei seiner Datenquellen bis zu 15 Bits verschieben, bevor die logische Operation durchgeführt wird. Das erlaubt eine Bewegung von Objekten nicht nur in Wortschrittweite, sondern »stufenlos«, sprich pixelweise.

- Der Blitter kann für jede Datenquelle und für das Datenziel verschiedene Moduli (Mehrzahl von Modulus!) verwenden. Dadurch können verschieden große Datenquellen verknüpft werden (was auch wieder den BOBs zugute kommt, da diese dann keine feste Größe haben müssen).

- Maskierung: Der Blitter kann das linke beziehungsweise rechte Datenwort der Blitterdaten »maskieren«, also Teile einer Linie quasi ausblenden. Das erlaubt logische Operationen ohne auf rechteckige wortbreite Daten beschränkt zu sein. Das wird im übrigen ebenfalls von den Betriebssystemroutinen für die BOBs (Blitter Objects) genutzt; BOBs können durch dieses Maskieren von Daten innerhalb einer Linie eine beliebige Form haben und müssen nicht in ein rechteckiges Worte-Schema passen.

- Linien ziehen: Mit dem Blitter können wir nicht nur Linien jeden Steigungswinkels ziehen, sondern der Linie auch ein Muster zuteilen, mit dem sie gemalt wird. Der Blitter kann außerdem spezielle Linien ziehen mit nur einem Pixel pro Bildschirmzeile (ein spezieller Modus, der während der Blitter-Flächenfüll-Operation benötigt wird).

- Grafikbereiche füllen: Der Blitter kann Flächen zwischen vorher gemalten Linien füllen.

- Nullerkennung (»Zero Detection«): Der Blitter kann nach einer logischen Operation herausfinden, ob irgendwelche gesetzten Bits

(1) als Resultat der Operation herausgekommen sind. Diese Möglichkeit kann genutzt werden für hardwareunterstützte Kollisionserkennung (zum Beispiel von der Betriebssytemroutine »DoCollision«).

Die einzelnen Blitter-Fähigkeiten werden in den folgenden Abschnitten zwar jede für sich besprochen, bauen aber aufeinander auf. Aus diesem Grunde ist es für das Verständnis wichtig, die folgenden Informationen der Reihe nach zu lesen. Die kurze Zusammenfassung oben wird ebenfalls zum Verständnis benötigt!

8.1 Der Datenschaufler

Der Hauptverwendungszweck des Blitters ist das Transferieren von großen Datenblöcken innerhalb des Speichers. Deswegen wurde der Blitter auch »Blitter« für »Block Image Transferer« genannt. Da diese Einheit jedoch noch weit mehr Fähigkeiten hat als nur den Datentransfer, wird sie von einem der Custom-Chip-Entwickler, Jay Miner, auch »Bimmer« für »Bit Image Manipulator« genannt.

Wie Bilder als Bitmap gespeichert sind, haben wir bereits zu Beginn des Hardwareteils besprochen. Der Blitter kann nun solche Bitmaps als Datenblöcke kopieren. Alles, was wir ihm dazu sagen müssen, ist die Startadresse, die Größe des Datenblocks und die Zieladresse. Die Daten werden Wort für Wort per DMA kopiert. Da die Übertragung wortweise geschieht, muß so ein Datenblock also immer an einer geraden Speicheradresse beginnen. Müssen wir ein Playfield kopieren, muß für jede einzelne Bitplane des Playfields eine solche Kopieroperation durchgeführt werden. Wenn der Transfer beendet ist, signalisiert der Blitter dem Prozessor durch ein Flag und einen Interrupt, daß der 68000 wieder darauf zugreifen darf.

So kann zum Beispiel schnell direkt in das auf dem Bildschirm sichtbare RAM geschrieben werden. Die Datenkopierfunktion wird vom Betriebssytem zum Beispiel auch für das sogenannte Double-Buffering verwendet: Zwei Playfields, die fast identisch sind, werden gleichzeitig im Speicher gehalten. Die Grafik wird dann in dem gerade nicht sichtbaren Playfield aufgebaut; anschließend wird erst auf das geänderte Bild umgeschaltet. Direkt danach wird das gesamte neue Playfield in das jetzt nicht sichtbare Playfield kopiert

(mit dem Blitter). Das hat zur Folge, daß ein schneller Grafikaufbau ohne Flackern des Bildes möglich ist.

8.2 Mehrere Datenquellen

Der Blitter kann mit bis zu vier DMA-Kanälen verwendet werden. Drei der DMA-Kanäle werden verwendet, um die Daten, die der Blitter benutzt, aus dem Speicher zu holen. Sie werden bezeichnet als Quelle A, B und C. Der vierte DMA-Kanal wird für die Ausgabe auf den Zielspeicher benutzt und wird Ziel D genannt. Wie Sie auf den folgenden Seiten sehen werden, ist es nicht immer notwendig, alle drei Quellen zu verwenden. Ebenso ist es nicht immer richtig, den Ziel-DMA-Kanal D zu benutzen.

Jeder Kanal kann unabhängig von den anderen an- oder ausgeschaltet werden durch Setzen bzw. Löschen der Bits 11, 10, 9 und 8 des Blitter-Registers BLTCON0 (Blitter Control 0). Diese vier Bits werden USEA, USEB, USEC und USED genannt. Alle drei Quellen (wenn angeschaltet) holen die Daten vom Speicher in serieller Folge (Wort für Wort) und schreiben sie in das Register. Innerhalb dieser Register des Blitters werden dann die logischen Operationen durchgeführt. Erst dann wird das Ergebnis an den DMA-Kanal für D übergeben, der es in die Zielspeicheradresse schreibt.

8.3 Die Datenverknüpfung

Die Fähigkeit, Datenbits von verschiedenen Bildquellen während des Datenkopierens zu verknüpfen, prädestiniert den Blitter zur schnellen Animation. So könnte man zum Beispiel ein rechteckiges Objekt mit einem existierenden Grafikobjekt auf dem Bildschirm verknüpfen (zum Beispiel ein Auto, das vor ein paar Häusern vorbeifahren soll).

Nehmen wir das Beispiel von Hintergrund und Auto, das sich auf diesem Hintergrund bewegen soll: Um das Auto zu animieren (es vor den Gebäuden fahren zu lassen), kopieren wir mit dem Blitter zuerst den Hintergrundbereich, auf den das Auto aufkopiert werden soll. Dann kopieren wir das Auto in seine erste Position. Anschließend müssen wir die gespeicherten Hintergrundinformationen wieder zurückkopieren, das Auto auf seine neue Position (1 Pixel weiter

rechts) kopieren und so weiter. Damit können wir eine ständig fortlaufende Sequenz von Hintergrund sichern, Objekt in Grafik kopieren und gesicherten Hintergrund neu an seine alte Position kopieren.

Nehmen wir einmal an, Quelle A ist der Umriß (Maske) des Autos, Quelle B ist eine der Autobitplanes (Übereinanderlagern von so vielen Planes wie die Hintergrundgrafik!) und Quelle C ist der Hintergrund (also die Gebäude). Folgende Operation sichert den Hintergrund, auf den das Auto kopiert werden soll (Ziel links, Quellen rechts):

T = AC

Diese Gleichung entspricht den Richtlinien der Originaldokumentation, anhand der wir dieses Beispiel einmal weiterspinnen wollen. Die Gleichung soll aussagen, daß der Hintergrund (C) in eine temporäre Zieladresse (T) kopiert (gesichert) werden soll, wo die Autoumriß-Maske (A) und der Hintergrund (C) zusammen existieren. Wir kopieren damit also genau den Teil des Hintergrundes, der nachher durch das Auto überlappt wird.

Nun kopieren wir das Auto mit der folgenden Operation in den Hintergrund:

C = AB + AC

Diese Gleichung sagt aus, daß das Ziel das gleiche ist wie die Hintergrundquelle (C), und daß der Hintergrund (C) durch die Autodaten (B) ersetzt werden soll, wo die Automaske (A) »wahr« ist, aber Hintergrund bleiben soll, wo die Maske nicht wahr ist. Das heißt, es wird nicht ein rechteckiger Speicherblock in den Hintergrund kopiert, sondern nur die Form des Autos (alles um das Auto herum wurde durch das Maskieren auf Null gesetzt und bei der Verknüpfung ignoriert). Innerhalb des kopierten Speicherblocks bleiben dann die Daten um das Auto herum so wie sie sind, da die durch das Maskieren erzeugten Nullbits mit der »und«-Verknüpfung mit dem Hintergrund zusammen nur den Hintergrund ergeben.

Jetzt müssen wir den Hintergrund wieder zurücksetzen (damit an der Stelle, wo jetzt das Auto ist, wieder der Hintergrund erscheint):

C = AT

Diese Gleichung sagt, daß der Hintergrund C mit dem gespeicherten Hintergrund T ersetzt werden soll, und zwar an der Stelle, wo die

Automaske (A) existiert. Das heißt, in logischen Operationen gesprochen, A »und« T ergibt C.

Durch das Verschieben der Autodaten (B) und der Autoform-Maske (A) an eine neue Position ~~und die fortschreitende~~ Wiederholung der drei obengenannten Schritte, erscheint das Auto als bewegtes Objekt über dem Hintergrund, der sich anscheinend selbst nicht bewegt (obwohl Daten hin- und hergeschoben werden). Die folgende Grafik verdeutlicht noch einmal, was unter den einzelnen Quellen und Zielen (A, B, C und T) zu verstehen ist.

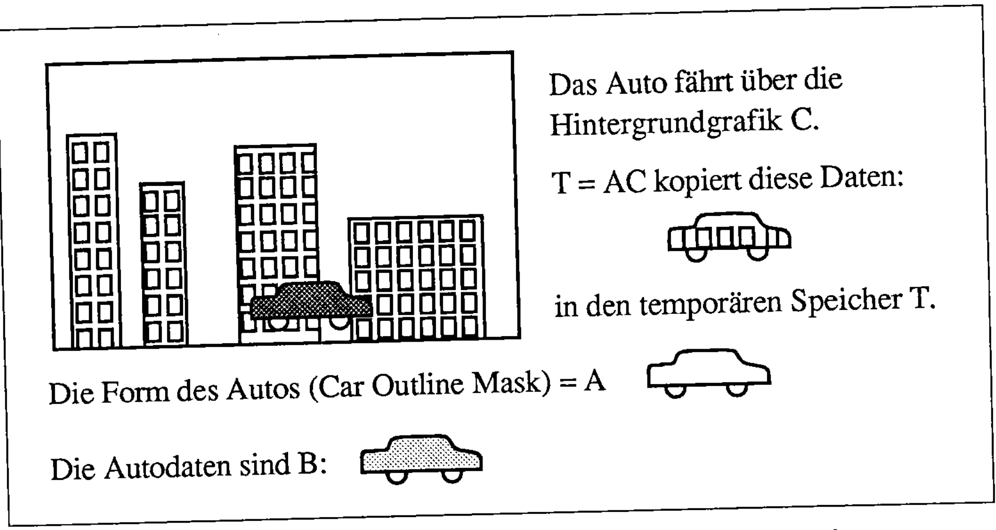


Bild 8.1: Die Fahrt eines Autos als Folge logischer Speicherverknüpfungen

8.3.1 Logische Kombinationen mit »Minterms«

Der Blitter vollführt verschiedene logische Operationen wie die vorhin genannte durch Kombinieren sogenannter »Minterms«. Ein Minterm (steht für Minimalausdruck einer Gleichung) ist eine von 8 möglichen logischen Kombinationen von Datenbits aus drei verschiedenen Datenquellen. Die folgende Gleichung benutzt zum Beispiel zwei Minterms, ABC und ABC

D = ABC + ABC

Das bedeutet, daß der logische Wert von D eine 1 ist, wenn entweder die links vom Pluszeichen stehende Kombination ABC gleich 1 ist oder die rechs vom Pluszeichen stehende Kombination 1 ist.

Eine andere Möglichkeit, diese Gleichung zu lesen, ist, daß D »wahr« ist, wenn sowohl A als auch B »wahr« sind (Zum Begriff »wahr« sollten Sie wissen, daß logische Operationen zwei Ergebnisse haben können: TRUE und FALSE, ins Deutsche mit »Wahr« und »Unwahr« übersetzt). Das ist der Fall, weil wir die Gleichung auch in folgender Form schreiben können:

D = AB (C + C)

Da jedoch der Term (C + C) immer wahr ist, können wir die Gleichung reduzieren auf D = AB. Aus diesem Grund gibt die Wahl der Minterm ABC und ABC die logische Operation D = AB. Diese zwei Minterms werden mit Bits 7 und 6 des Registers BLTCON0 angewählt.

Die Minterms können durch die BLTCON0-Kontrollbits LF7 bis LF0 wie folgt angewählt werden:

MINTERMS:	ABC	ABC	ABC	ABC	ABC	ABC	ABC	ABC
BLTCON0-Bits:	7	6	5	4	3	2	1	0

Da es 8 Minterms gibt, können 256 verschiedene Gleichungsmöglichkeiten angewählt werden. Die logische Kombination der Quellen geht nach Setzen dieser Bits automatisch während des Kopierens vor sich wie bereits weiter oben beschrieben.

Die Kombination der einzelnen Bits von BLTCON0 ergibt einen Byte-Wert. Der Bytewert der 8-Minterm-Bits wird LF-Code genannt. Die Kombination D = AB + AC ergibt zum Beispiel einen LF-Wert von $CA. CA wird von den Entwicklern des Amiga »cookie cut minterm selector« genannt. »Cookie Cut« heißt soviel wie »Plätzchen aus dem Teig ausstechen«. Das soll bedeuten, daß wir mit dem Minterm LF-Code CA bestimmte Formen aus einer Hintergrundgrafik herauskopieren können, wie am Anfang dieses Abschnitts anhand des Beispiels mit dem fahrenden Auto erklärt wurde.

8.4 Mehrfach-Moduli – verschieden große Datenquellen

Wie auch in der Playfield-Hardware wird im Blitter ein sogenannter Modulus verwendet, um verschieden große Bilder bearbeiten zu können. Beim Blitter behandeln wir mit dem Modulus kleinere Datenblöcke innerhalb von größeren Datenblöcken. Hier ist der Modulus die Differenz zwischen der Breite des kleineren (den wir manipulieren wollen) und des größeren Datenblocks. Der Blitter verfügt über insgesamt vier Moduli – damit können die drei Datenquellen und das Datenziel jeweils eine verschieden große Bitplanegröße haben.

Nehmen wir als Beispiel hierfür eine Bitplane, die groß ist. Innerhalb dieser Bitplane befindet sich nun ein kleineres Window. Der linke Rand des kleineren Windows ist genau zwei Bits vom linken Rand der großen Bitplane entfernt, der rechte Rand ist ebenso zwei Bits vom rechten Rand der größeren Plane entfernt. Unsere Operationen wollen wir mit dem kleineren Quell-Window aus der großen Bitplane vollführen. Um in dem kleineren Datenblock zu operieren, muß die Adressenreihenfolge jetzt also beim ersten Bit des kleinen Windows beginnen, aufsteigend bis zum rechten Rand des Windows fortgeführt werden, und schließlich mit 2 Bits Pause dazwischen erst bei der nächsten Zeile beginnen. Das erfordert einen Inkrement (+2) und am Ende jeder Windowzeile die Addition des Sprungwertes 4, damit der Adreßzeiger auf den Start der nächsten Zeile zeigt. Dieser Sprungwert wird – wie alle Sprungwerte der Amiga-Custom-Chips – Modulus genannt. Er entspricht dem Unterschied zwischen der Breite der größeren Bitplane und der Breite des kleineren Bildteiles. Der Modulus wird als Word in sein Register geschrieben. Alles, was nicht in 16-Pixel-Schritten voneinander entfernt ist, muß durch Maskieren der fehlenden Bits zum 16er-Schritt aufgerechnet werden.

Beachten Sie bitte, daß die Hardware zwar mit Worten für Moduli und Adreßzeiger arbeitet, jedoch die Werte, die in diese Hardwareregister gelesen werden, vom 68000 als Bytes verwertet werden. Zum Beispiel würde ein Sprungwert 4 für einen Modulus in Wirklichkeit eine 8 sein, wenn er vom 68000 in das Register geschrieben wird.

8.5 Es geht auch andersherum – absteigende Adressierung

Es ist sehr wichtig, die Adressierungsrichtung (Inkrement oder Dekrement) kontrollieren zu können, wenn sich der Quellbereich und der Zielbereich überlappen. Durch Festlegen aufsteigender oder absteigender Reihenfolge des Datenkopierens kann man den Blitter davon abhalten, Quelldaten während des Kopierens der überlappten Daten zu zerstören.

Wollen wir zum Beispiel Daten in eine höhere Speicheradresse kopieren, wobei sich aber die Quelle und das Ziel überlappen, müssen wir die absteigende Reihenfolge (Dekrement) Methode verwenden. Sollen Daten in einen niedrigeren Adreßbereich geschaufelt werden, wobei sich Quelle und Ziel überlappen, müssen wir die Inkrement-Methode (aufsteigende Reihenfolge) benutzen. Im Normalfall benutzt der Blitter die Inkrement-Methode. Die absteigende Adressierung wird durch Setzen des Bits 1 im Register BLTCON1 aktiviert.

8.6 Bitweise Datenverschiebungen

Der Blitter enthält eine Schaltung, die als »Barrel Shifter« bekannt ist. Der Barrel Shifter kann mit Datenquelle A und Datenquelle B verwendet werden. »Shifting« heißt nichts anderes als Verschiebung; dieser Shifter erlaubt Bewegung von Daten auf Pixelbasis, obwohl die Grafiken nur als ganze 16-Bit-Worte der Grafik adressiert werden.

Wie vorhin unter »Logische Operationen« erklärt, benötigt die Bewegung eines Objekts wie das unseres Autos (B) über einen Hintergrund (C) sowohl die Bitdaten des Autos (B) als auch die »Auto-Maske«, also quasi den Umriß des Autos (A). In unserem Fall müssen beide jedesmal um ein Bit verschoben (ge»shiftet«) werden, wenn der Hintergrund in den temporären Speicher gerettet wird (T = AC), das Auto auf den Hintergrund plaziert wird (LF-Code CA, »Cookie Cut«) und der Hintergrund zurückkopiert wird (C = AT). Das klingt zwar kompliziert, aber so schlimm ist es auch wieder nicht.

Insgesamt gibt es zwei sogenannte Shift-Kontrollen. Die Bits 15 bis 12 des BLTCON0-Registers wählen den Shift-Wert für Quelle A. Die Bits 15 bis 12 des BLTCON1-Registers wählen den Shift-Wert für Quelle B. Beide Werte sind im Normalfall identisch. Durch Erhöhen des Shift-Wertes ist also jedesmal das Auto ein Pixel weiter rechts, obwohl es im Speicher konstant ist. Durch das Shiften verschieben wir lediglich die Daten innerhalb des Blitters, bevor er damit seine logischen Operationen durchführt. Das ruft den Eindruck einer pixelweisen Bewegung hervor. Erst, wenn wir diesen Vorgang 16 mal durchgeführt haben, kopieren wir das Auto ein Wort weiter rechts in die Grafik, wo wir wieder mit einem Shift-Wert von 0 anfangen und bis 15 aufwärts zählen. Damit können wir ein Objekt stufenlos über den gesamten Bildschirm bewegen. Das macht sich auch das Betriebssystem in seinen Animationsroutinen zunutze.

8.7 Maskierung von Daten

Wenn die Pixelbreite eines Objektes nicht genau ein Vielfaches von 16 ist, kann der Blitter entweder das linke oder das rechte Ende »abschneiden«, um nur mit den wirklichen Bitgrenzen anstatt von wortbreiten Rechtecken arbeiten zu können. Das Maskieren des ersten bzw. des letzten Wortes ist sinnvoll, wenn es notwendig ist, einen Zeichensatz in nah aneinandergepackter Organisation im Speicher zu halten (also ohne Zwischenräume), zum Beispiel wenn wir die Bitimages der Buchstaben I und H beide nebeneinander mit einer Gesamtbreite von 16 Bits im Speicher liegen.

Um das I-Zeichen aus diesem 16 Pixel breiten Muster zu isolieren, müssen die ersten 11 Bits des Rechtecks maskiert werden. Der Blitter hat diese Fähigkeit, genannt First-Word-Mask, und verwendet sie beim am weitesten links liegenden Wort in jeder Linie (Natürlich nur, wenn wir ihm das gesagt haben). Nur wenn ein 1-Bit in der Maske des ersten Wortes ist, wird dieses Bit von Quelle A in der logischen Operation auftauchen. Durch das Maskieren »übersieht« der Blitter bei den logischen Operationen alle Bits des ersten Worts, an deren Stelle in der Maske eine 0 steht.

Genauso, wie der Blitter diese »First Word Mask« verwendet, um die linke Seite der Daten aus Quelle A zu maskieren, kann er auch das letzte Wort maskieren. Diese Maske heißt BLTALWM (Blitter A Last Word Mask); damit wird das am weitesten rechts liegende

Wort der Daten aus Quelle A maskiert. Dadurch ist es möglich, einen rechteckigen Datenblock aus einer Datenquelle zu verarbeiten, dessen rechter Rand nicht auf eine Wortgrenze fällt.

Wenn die Datenquelle (das Datenwindow) nur ein Wort breit ist, werden sich die First-Word- und Last-Word-Maske überlappen. Dabei werden die Daten aus Quelle A nur an den Blitter weitergegeben, wo beide Masken den Zustand »wahr« (TRUE) haben. Deshalb sollte man nicht benötigte Masken immer mit $FFFF (16 wahren Bits) füllen.

8.8 Nullerkennung – Kollision durch Hardware festgestellt

Der Blitter verfügt über ein sogenanntes Null-Flag, das getestet werden kann, um herauszufinden, ob die logische Operation eine Null als Resultat hat. Dieses Nullflag (BZERO) in Bit 13 des DMACONR-Registers wird »wahr«, also gesetzt sein, wenn bei einer logischen Operation nur Nullen als Ergebnis herausgekommen sind.

Diese Fähigkeit wird normalerweise benutzt, um Kollisionen zwischen Software-Objekten (BOBs) feststellen zu können: Die Objekte werden durch logisch »und« verknüpft und auf Überlappen getestet. Die Operation D = AB wird durchgeführt (D kann abgeschaltet werden), und wenn A und B sich nicht überlappen, wird das Zero-Flag auf 1 (TRUE) gesetzt sein.

8.9 Füllen von Flächen mit dem Blitter

Daß der Blitter sehr schnell beim Füllen von Flächen sein kann, werden Sie sicher schon des öfteren gemerkt haben. Das Flächenfüllen kann im Blitter sogar gleichzeitig mit dem Transferieren und Verknüpfen von Daten vor sich gehen, da andere DMA-Kanäle dafür benutzt werden.

Die Füll-Operation hat jedoch eine grobe Beschränkung: Die zu füllende Fläche muß durch das Zeichnen von speziellen Linien, die nur ein Bit breit sein dürfen, definiert werden. Für das Zeichnen solcher Linien existiert natürlich ein spezieller Linienmodus.

8.9.1 Normales (»inklusives«) Flächenfüllen

Ein typischer Areafill mit dem Blitter füllt bespielsweise eine Fläche, deren linke und rechte Grenzen durch zwei senkrechte Linien begrenzt sind. Um eine solche Fülloperation durchzuführen, müssen zuerst mit dem Blitter zwei vertikale Linien gezogen werden, jede 1 Pixel breit. Danach setzen wir den Modulus auf die Breite des gesamten Bildes minus der Breite der zu füllenden rechteckigen Fläche (Register BLTxMOD mit x = A, B, C oder D). Wir setzen die Quell- und Zielzeiger des Blitters auf denselben Wert. In diesem Fall wird nur eine Quelle und ein Ziel benötigt. Diese Zeiger sollten auf das letzte Wort (also das untere rechte Wort) des rechteckigen Bereichs zeigen (BLTxPTH, BLTxPTL mit x = A, B, C oder D). Jetzt setzen wir den Blitter auf absteigende Adressierung. Die Füll-Operation funktioniert nur bei absteigender Reihenfolge korrekt (BLTCON1, Bit 1 = 1). Wir verwenden das Kontrollbit »FCI« (für Fill-Carry-In), um zu definieren, wie die Fülloperation vor sich gehen soll (BLTCON1, Bit 2 = 0). Das definiert den Füll-Start-Status als eine Null. Nun definieren wir noch die horizontale und vertikale Größe des Rechtecks, das die zu füllende Grafik »einrahmt«. Dieser Wert muß in das Größenkontrollregister (BLTSIZE) geschrieben werden, um damit schließlich das Füllen zu starten.

Der Blitter benutzt das FCI (Fill-Carry-In) Bit als den Startstatus des Füllens (das heißt was an der rechten Ecke einer jeden Linie ist). Für jedes »1«-Bit im Quellbereich dreht der Blitter den Füllstartstatus um und füllt den Teil der Grafik entweder mit Einsen oder nicht. Das geht für jede Zeile von unten nach oben so, bis die linke Ecke der BLTSIZE erreicht ist. An diesem Punkt stoppt der Füllvorgang. Für ein weiteres Beispiel bezüglich des FCI-Bits sehen Sie sich bitte die nächste Seite an.

Bitte beachten Sie, daß eine Fülloperation auch während anderer Blittertransfer-Operationen vorgenommen werden kann. Trotzdem werden beide Vorgänge oft einzeln durchgeführt, so auch in unserem obengenannten Beispiel.

8.9.2 »Exklusives« Füllen eines Grafikbereichs

Im Register BLTCON1 gibt es zwei sogenannte »Fill Enable Bits«. Die beiden Bits heißen IFE = Inclusive Fill Enable und EFE = Ex-

clusive Fill Enable. IFE bedeutet, daß bei bei dem Füllprozeß die Umrandung (die beiden 1 Pixel breiten Linien) unverändert gelassen werden. EFE bedeutet, daß beim Füllen die Umrahmung auf der linken Seite beseitigt (ebenfalls gefüllt) wird.

8.10 Das Linienziehen

Der Blitter kann Linien ziehen, die innerhalb eines Koordinatensystems liegen, das von (0,0) bis (1024,1024) geht, also über den Bereich der normalen Grafikauflösung hinaus. Das »Lines«-Demo auf der Workbench 1.2-Diskette zeigt das recht eindrucksvoll.

Der Blitter wird in den »Line Draw«-Modus geschaltet, indem eine 1 in Bit 0 des BLTCON1-Registers geschrieben wird. Das erzeugt eine Neudefinition einiger anderer Kontrollbits in BLTCON0 und BLTCON1. Linien können normal oder invers gezogen werden, aber auch durchgehend oder mit Mustern versehen sein. Spezielle Linien mit nur einem Punkt pro Rasterstrahlzeile für Benutzung mit Areafill sind ebenfalls möglich.

Viele der Blitter-Register haben im Linedraw-Modus andere Funktionen als im Normalfall. Die neuen Funktionen der Register sind in untenstehender Tabelle genannt.

Register	Bit	Bitname	Status	Zweck
BLTCON1	0	LINE	1	Anschalten des Linedraw-Modus
BLTCON1	15 - 12	BSH	0	Muster-Start bei Bit 0
BLTCON1	1	SING	0,1	Auf 1-Pixel-weite Linie schalten
BLTCON0	15 - 12	START		X- Position des ersten Pixels
BLTCON0	11 - 8	USE	1011	Für Linienziehen benötigt
BLTADAT	Alle		8000	Für Linienziehen nötiger Index
BLTBDAT	Alle			Linien-Muster-Register
BLTSIZE	5 - 0			Für Lnienziehen benötigt
BLTSIZE	15 - 6			Linienlänge (bis 1024)
BLTAMOD	Alle			2(2y-2x) *
BLTBMOD	Alle			2(2y) *
BLTCMOD	Alle			Breite des gesamten Bildes
BLTDMOD	Alle			Breite des gesamten Bildes
BLTAPTL	Alle			(2y-x) *
BLTCPT	Alle			Startadresse der Linie
BLTDPT	Alle			Startadresse der Linie
BLTCON1	4, 3, 2			Oktant Select Code (siehe unten)

Tabelle 8.1: Die Bits des Linedrawmodus

Wie Sie sehen, werden hier eine ganze Menge Register zweckentfremdet. Ist der Blitter im Linedraw-Modus, kann er für andere Zwecke nicht verwendet werden. Für die mit * gekennzeichneten Register gilt: x und y sind die Höhe und Breite des kleinsten Rechtecks, das die Linie einschließt.

Eine Linie auf diese Weise zu ziehen, ist nun allerdings recht kompliziert. Obwohl dieser Buchteil eigentlich zeigen soll, wie man die Hardware auch direkt ansprechen kann, empfehle ich persönlich die Betriebssystemroutinen für so komplizierte Dinge. Für diejenigen, die trotzdem Linien durch direktes Ansprechen der Register ziehen wollen, hier noch das Schema, das den sogenannten »Octant Select Code« beschreibt:

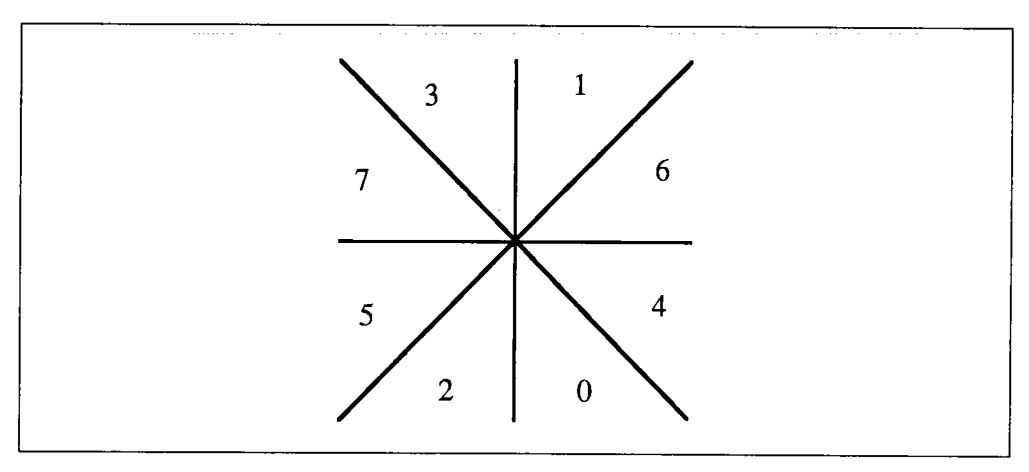


Bild 8.2: Codes für das Linienziehen

Der Blitter benötigt den Oktanten, um die Linie richtig berechnen zu können. Beim Zeichnen einer Linie darf kein einziges der obengenannten Register falsch gesetzt werden!

Auf eine Zusammenfassung der Blitter-Register habe ich bewußt verzichtet, um nicht unnötige Verwirrung zu stiften. Wer trotzdem Bescheid wissen möchte, betrachte bitte den Anhang »Register«.



9

Der Copper

Dafür, daß bei jedem Bildaufbau auch alle Daten richtig erscheinen, ist in hohem Maße der Copper mitverantwortlich. Durch Befehle an diesen Coprozessor werden nämlich bei jedem Aufbau des Bildes die Adreßregister für Bitplanes und Sprites wieder aktualisiert. So muß sich nicht der Hauptprozessor darum kümmern, wann welche Teile des Speichers über die Videologik an den Rasterstrahl übergeben werden.

Der Copper ist ein Allzweck-Coprozessor, der im Amiga-Customchip Agnus enthalten ist. Seine Instruktionen holt er per DMA aus dem RAM. Der Copper kann direkt auf alle Register der Custom-Chips zugreifen. Dadurch eignet er sich hervorragend, um zum Beispiel in der Mitte des Bildschirms das Display völlig zu ändern (was von Intuition für die sogenannten Screens genutzt wird). Der Copper erledigt aber auch – wie bereits gesagt – die gesamte Arbeit des Zurücksetzens der Register, die während des vertikalen Blanking geändert werden müssen. Wie Sie bereits in den vorangegangenen Abschnitten gesehen haben, ist diese Wiederauffrischung der Register unbedingt notwendig, um Playfields und Sprites anzuzeigen. Man kann aber auch die Farbpalette irgendwo auf dem Bild ändern und sogar den Blitter kontrollieren.

Eine der Fähigkeiten des Coppers ist es, auf eine bestimmte Rasterstrahlposition zu warten und dann Daten in ein Systemregister zu schreiben (Befehle WAIT und MOVE). Während der WAIT-Periode »untersucht« der Copper den Inhalt des Rasterstrahlzählers direkt. Das bedeutet, daß der Datenbus frei für den 68000 oder die anderen

Elemente der Custom-Chips ist, während der Copper auf diese Rasterstrahlposition wartet.

Wenn die Wait-Bedingung erfüllt ist, »stiehlt« der Copper Taktzyklen von entweder dem Blitter oder dem 68000, um die gewünschte Information in das gewählte Register zu schreiben. Der Copper ist aber ein 2-Zyklus-Prozessor, der den Bus nur während der ungeraden Speicherzugriffszyklen benötigt. Das vermeidet Kollision des Coppers mit Audio, Disk, Refresh, Sprites und den meisten Low-Resolution-Display DMA-Zugriffen, da diese alle die geradzahligen Speicherzyklen verwenden. Deswegen benötigt der Copper nur Priorität über den 68000 und den Blitter.

Wie bei allen anderen DMA-Kanälen des Amiga kann auch der Copper Daten beziehungsweise Instruktionen nur aus den unteren 512KByte Speicher holen.

9.1 Befehle zur Registermanipulation

Als Coprozessor hat der Copper sein eigenes Befehlsset; er verfügt nur über drei Befehle, was man aber mit diesen Befehlen machen kann, ist eine ganze Menge:

WAIT:
Warten auf eine bestimmte Rasterstrahlposition (angegeben als x und y-Koordinaten).

MOVE:
Kopieren eines Wertes in eines der Custom-Chip-Register.

SKIP:
Überspringen des nächsten Befehls, wenn der Rasterstrahl bereits eine bestimmte Bildschirmposition erreicht oder überschritten hat.

Alle Copper-Befehle bestehen aus zwei 16-Bit-Worten in aufeinanderfolgenden Speicheradressen. Jedesmal, wenn sich der Copper einen Befehl aus dem RAM holt, holt er beide Worte.

Obwohl der Copper direkt nur auf die Register der Custom-Chips zugreifen kann, kann er den gesamten Speicher der unteren 512 KBytes RAM dadurch manipulieren, indem er Blitter-Operationen durchführt (der Copper greift auf den Blitter zu, der wiederum auf das RAM zugreift).

Die Befehle WAIT und MOVE werden nachstehend noch näher beschrieben. Der SKIP-Befehl folgt in einem späteren Abschnitt, da er zu den fortgeschrittenen Themen der Copperprogrammierung zählt.

9.1.1 Register verändern

Der Move-Befehl transferiert Daten aus dem RAM in ein Zielregister. Die zu transferierende Information ist im zweiten Wort des MOVE-Befehls enthalten; das erste Wort beinhaltet die Adresse des Zielregisters. Der Move-Befehl setzt sich aus den zwei Befehlsworten IR1 und IR2 zusammen:

Erstes Befehlswort (IR1):

Bit 0 Immer auf 0 gesetzt.

Bits 8 - 1 Register-Zieladresse (DA8-1).

Bits 15 - 9 Nicht benutzt, sollten auf 0 gesetzt werden.

Zweites Befehlswort (IR2):

Bits 15 - 0 16 Bits Daten, die in das Zielregister kopiert werden sollen.

Der Copper kann mit dem MOVE-Befehl in folgende Register speichern:

- Jedes Register, dessen Registeradresse $20 oder höher ist.

- Jedes Register, dessen Registeradresse zwischen $10 und $20 liegt, wenn das »Copper Danger Bit« (Gefahrenbit) auf 1 gesetzt ist. Das Danger-Bit ist im Kontrollregister COPCON enthalten.

- Der Copper kann nicht in Register schreiben, deren Registeradresse niedriger als $10 ist. (Bitte nicht die Registeradressen mit Speicheradressen verwechseln!)

9.1.2 Der Wait-Befehl

Der Wait-Befehl bringt den Copper dazu zu warten, bis die Rasterstrahlzähler gleich oder größer als die Koordinaten sind, die im Be-

fehl genannt wurden. Während des Wartens zieht sich der Copper vom Bus zurück und benötigt keine Speicherzyklen.

Das erste Befehlswort enthält die vertikalen und horizontalen Koordinaten der Rasterstrahlposition. Das zweite Wort enthält Bits, die verwendet werden, um eine Maske zu erstellen, die dem Copper sagen, welche Bits der Strahlposition für den Vergleich verwendet werden sollen.

Erstes Befehlswort:

Bit 0 Immer auf 1 gesetzt.

Bits 15 - 8 Vertikale Position (genannt VP).

Bits 7 - 1 Horizontale Position (genannt HP).

Zweites Befehlswort:

Bit 0 Immer auf 0 gesetzt.

Bit 15 Das »Blitter finished Disable Bit«.

Bits 14 - 8 Vertikale Positionsvergleichsbits (genannt VE für Vertical Enable).

Bits 7 - 1 Horizontale Positionsvergleichsbits (genannt HE für Horizontal Enable).

9.1.3 Rasterstrahlpositionen

Die folgenden Informationen gelten gleichermaßen für den WAIT-Befehl und den SKIP-Befehl: Die horizontale Rasterstrahlposition kann einen Wert von $00 bis $E2 haben. Das niederstwertige Bit wird für den Vergleich nicht verwendet, so daß 113 Positionen für die Copper-Operationen verfügbar sind. Das entspricht vier Pixels in LoRes und 8 Pixels in HiRes. Wir können damit also alle vier Punkte eine Farbe ändern. Durch diesen kleinen Trick haben wir auch ohne den aufwendigen HAM-Modus mehr als 32 Farben gleichzeitig auf dem Bildschirm.

Das horizontale Blanking fällt in den Bereich von $0F bis $35. Der Standard-Bildschirm (320 Pixels breit) hat einen unbenutzten horizontalen Bereich von $04 bis $47 (während dessen »Anzeige« nur die Hintergrundfarbe auf den Bildschirm gebracht wird).

Die vertikale Rasterstrahlposition gibt die Zahl der Linie an, in der sich der Strahl befindet; sie hat einen maximalen Wert von 255. In Wirklichkeit gibt es aber 262 mögliche vertikale Positionen. Es kann zu kleinen Komplikationen kommen, wenn wir in diesen unteren 6 oder 7 Rasterstrahlzeilen etwas erscheinen lassen wollen.

9.2 Die Register des Copper

Der Copper verfügt über mehrere Arten von Registern: Die Adreßregister, die »Jump Adreß Strobes« und die Kontrollregister.

9.2.1 Befehlsadreßregister

Der Copper hat zwei Sets von Adreßregistern, die ihm sagen, wo im Speicher seine Befehlslisten liegen:

COP1LCH	Die 3 High Bits der ersten Copperlisten-Adresse.
COP1LCL	Die 16 Low Bits der ersten Copperlisten-Adresse.
COP2LCH	Die 3 High Bits der zweiten Copperlisten-Adresse.
COP2LCL	Die 16 Low Bits der zweiten Copperlisten-Adresse.

Trotz der hier nur benötigten 18-Bit-Information ist es auch hier möglich, ganze 32-Bit-Longwords in das Highword-Register zu schreiben.

Die Copper-Adreßregister beinhalten die zwei indirekten Sprungadressen, die vom Copper verwendet werden. Der Copper holt sich seine Daten durch die Verwendung seines Programmzählers und inkrementiert den Programmzähler nach jedem Datenholen. Wenn eine »Jump Adress Strobe« geschrieben wird, wird das entsprechende Adreßregister in den Programmzähler geladen. Das verursacht den Sprung des Coppers an eine neue Adresse (die aus dem Adreßregister), von der der nächste Copper-Befehl geholt wird. Von da aus geht der Zugriff im Speicher sequentiell vor sich, bis der Copper durch einen weiteren »Jump Adreß Strobe« unterbrochen wird.

Bitte beachten Sie, daß nach dem Start eines jeden vertikalen Blanking-Intervalls COP1LC automatisch verwendet wird, um den Programmzähler zu starten. Wenn das vertikale Blanking vor sich

geht, wird die Befehlsliste also einfach abgebrochen, um an der in COP1LC stehenden Adresse weiterzumachen.

9.2.2 Strobe-Adressen

Wenn wir in eine »Strobe«-Adresse schreiben, setzt der Copper seinen Programmzähler auf das entsprechende Adreßregister. Der Copper kann auch in seine eigenen Adreßregister schreiben, um programmierte Sprünge durchzuführen. So könnten wir beispielsweise eine Adresse mit dem MOVE-Befehl in COP2LC schreiben. Danach lädt jeder MOVE-Befehl, der COPJMP2 adressiert, diese indirekte Adresse in den Programmzähler.

Der Copper verfügt über zwei solche Sprungadressen:

COPJMP1	Wiederstarten der Copperbefehle bei der Adresse, die in COP1LC enthalten ist.
COPJMP2	Wiederstarten der Copperbefehle bei der Adresse, die in COP2LC enthalten ist.

9.2.3 Kontrollregister

Der Copper kann auf einige der Custom-Chip-Register immer zugreifen; einige andere können nur erreicht werden, wenn ein spezielles Kontrollbit auf 1 gesetzt ist. Und ein paar Register lassen sich überhaupt nicht manipulieren. Innerhalb der Gruppe, auf die der Copper nicht zugreifen kann, liegen seine eigenen Kontrollregister. Das bedeutet, daß es einige Prozessorzeit benötigt, den Copper erstmal dazu zu bringen, in andere Custom-Chip-Register zu schreiben.

Das Copperkontrollregister COPCON beinhaltet nur ein Bit, Bit 1. Dieses Bit, genannt CDANG für »Copper Danger Bit«, schützt alle Register zwischen $10 und $1F. Das beinhaltet auch die Blitterkontrollregister. Wenn CDANG 0 ist, kann in diese Register nicht durch den Copper geschrieben werden. Das Schützen dieser Register hält den Copper davon ab, durch eine falsche Befehlsliste den gesamten Systemspeicher (über den Blitter) zu zerstören. Nach einem Reset wird das CDANG-Bit automatisch gelöscht.

9.3 Die Zusammensetzung einer Copper-Befehlsliste

Eine Copper-Befehlsliste enthält das gesamte Zurücksetzen der Register während des vertikalen Blanking-Intervalls und die Änderungen, die mitten im Bildschirm vorgenommen werden müssen. Wenn wir planen, was während eines jeden Rasterstrahldurchlaufs passieren soll, ist es einfacher, jeden Aspekt des Bildes als separates Subsystem zu betrachten; also Playfields, Sprites, Audio, Interrupts und so weiter. So erstellen wir für jedes dieser Subsysteme eine einzelne Liste. Wenn alle diese Listen der zu erledigenden Copper-Jobs fertig sind, müssen wir daraus eine einzelne Copperliste zusammenbauen, die der Copper einmal pro Rasterstrahldurchlauf ausführt. Das Zusammensetzen der Copperlisten aus Einzelkomponenten wurde übrigens auch im Betriebssystem des Amiga realisiert (hierzu gibt es die Copperlist-Befehle, zum Beispiel *MrgCopList*).

Beachten Sie beim Erstellen der Copperliste unbedingt, daß die endgültige Liste in der Reihenfolge sein muß, in der der Rasterstrahl das Bild zeichnet. Der Rasterstrahl geht von Position (0,0) von links oben zeilenweise nach rechts unten bis (226,263). Ein Befehl, der etwas bei Position (0,100) macht, sollte also nach einem Befehl kommen, der das Bild bei Position (0,60) ändert. Arbeiten wir als Programmierer mit dem Betriebssystem, werden unsere Copperbefehle automatisch in die richtige Reihenfolge gebracht.

9.4 Der Copper legt los

Nach dem Anschalten oder nach einem Reset muß eines der Copperadreßregister (COP1LC oder COP2LC) initialisiert werden und in seine Strobe-Adresse geschrieben werden, bevor die Copper-DMA angeschaltet wird. Normalerweise wird COP1LC verwendet, weil dies das Register ist, daß bei jedem vertikalen Blanking wiederverwendet wird.

Nehmen wir an, wir haben bereits eine Copperliste an der Speicherstelle Coplist angelegt. Die folgende Sequenz von 68000-Befehlen dient dazu, den Copper ordnungsgemäß zu initialisieren, wenn diese Liste bei der Adresse CopList beginnt:

```
move.l      Coplist,a0
move.l      a0,COP1LCH
move.w      COPJMP1,d0
move.w      #SETBIT + COPPERDMA,d0
move.w      d0,DMACONW
```

Jetzt wird der Copper, wenn die Inhalte von COP1LC nicht verändert werden, immer wieder an derselben Stelle für jeden Rasterstrahldurchlauf beginnen. Das ergibt eine wiederholbare Schleife, die (wenn die Liste korrekt ist) das Bild stabil hält.

Im Normalfall übernimmt diese Arbeit bereits das Boot-ROM, das schon aktiviert wird, bevor überhaupt das eigentliche Betriebssystem geladen wird. Wir müssen jedoch für jedes Display, das wir anzeigen wollen, den Copper entsprechend initialisieren.

Der Copper verfügt über keinen Stop-Befehl. Um abzusichern, daß der Copper aufhört zu arbeiten und nichts tut, bis die Bildanzeige endet und der Programmzähler von vorne beginnt, sollte der letzte Befehl ein Wartebefehl sein, der eigentlich nie vorkommen kann. Ein typisches Beispiel ist, auf Vertikale Position = $FF und Horizontale Position = $FE zu warten. Ein HP von mehr als $E2 ist nicht möglich. Wenn die Bildanzeige beendet ist und das vertikale Blanking beginnt, wird der Copper automatisch wieder auf den Anfang seiner Befehlsliste gesetzt und dieser endgültige WAIT-Befehl kommt nie zum Ende.

Wir können den Copper auch stoppen, indem wir seine Fähigkeit, seine Befehle per DMA zu holen, einfach abschalten. Das Register DMACON kontrolliert alle DMA-Kanäle. Bit 7, COPEN (Copper Enable) muß auf 0 gesetzt werden, wenn der Copper gestoppt werden soll.

9.5 Der Skip-Befehl

Der Skip-Befehl bringt den Copper dazu, den nächsten Befehl zu überspringen, wenn die Rasterstrahlzähler gleich oder größer dem Wert sind, den wir im Befehl angegeben haben.

Die Inhalte der SKIP-Befehlsworte sind identisch zum Wait-Befehl, bis auf Bit 0 des zweiten Wortes. Dieses muß hier gleich 1 sein, um das Befehlswort als SKIP-Befehl zu identifizieren.

9.6 Der Copper im Interlace-Modus

Im Interlace-Modus durchläuft der Rasterstrahl den Bildschirm zweimal von oben nach unten. Während des ersten Durchlaufs werden die Zeilen mit ungeraden Nummern angezeigt, beim zweiten Durchlauf die Zeilen mit geraden Nummern. Beim zweiten Durchlauf passiert das Anzeigen des zweiten Playfields jedoch um eine halbe Rasterstrahlzeile verschoben. Ein Interlace-Bild ist im Speicher abgelegt, wie eine einzelne Grafik, wird jedoch wie zwei separate Bilder angezeigt.

Die Startadresse des geraden Feldes ist eins höher als die des ungeraden Feldes. Deswegen muß der Bitplane-Adreßzeiger für jedes der beiden Felder unterschiedliche Werte haben. Das bedeutet, daß zwei separate Copperlisten benötigt werden, eine für das ungerade Playfield und eine für das gerade Playfield.

Damit der Copper immer die richtige Liste ausführt, senden wir einen Interrupt an den 68000 gleich nach der ersten angezeigten Zeile. Wenn der Interrupt ausgeführt ist, ändern wir die Werte des COP1LC-Registers, das damit auf die zweite Liste zeigt. Dann, nach dem vertikalen Blanking, wird COP1LC automatisch zurückgesetzt, um auf die erste Liste zu zeigen.

Für mehr Information zum Interlace-Modus schlagen Sie bitte im Abschnitt über Playfield-Hardware nach.

9.7 Blitter-Kontrolle durch den Copper

Wenn der Copper benutzt wird, um eine Reihe von Blitter-Operationen durchzuführen, muß er zuerst auf den sogenannten »Blitter Finished Interrupt« warten. Dieser Interrupt ist ein Signal, das der Blitter aussendet, um zu sagen »ich bin fertig, jetzt kannst du mich benutzen«. Ein Ändern der Blitter-Register, während der Blitter gerade arbeitet, bringt nämlich nicht voraussehbare Ergebnisse. Für diesen Zweck beinhaltet der WAIT-Befehl ein zusätzliches Kon-

trollbit, auch BFD für »Blitter Finished Disable« genannt. Im Normalfall ist dieses Bit eine 1, und nur der Rasterstrahl-Positionsvergleich kontrolliert den WAIT-Befehl.

Wenn das BFD-Bit eine 0 ist, wird die Logik des WAIT-Befehls geändert. Hier wird der Copper warten, bis der Rasterstrahlpositionsvergleich »wahr« ist und der Blitter fertig ist. Der Blitter ist mit seiner Arbeit fertig, wenn das Blitter-Finish-Flag gesetzt ist (was durch den vom Blitter erzeugten Interrupt gemacht wird). Ein Löschen dieses Bits sollte mit allergrößter Vorsicht gehandhabt werden. Es könnte womöglich einige Bilder oder Objekte daran hindern, korrekt auf dem Bild zu erscheinen.

10

Die Kontrollhardware

Dieser Abschnitt beschreibt die Kontrollhardware des Amiga-Systems. Die folgenden Themen fallen in dieses Sachgebiet:

- Wie Playfield-Prioritäten relativ zu Sprites erstellt werden.
- Wie die Kollision zwischen Objekten erfaßt wird.
- Wie der direkte Speicherzugriff (DMA) kontrolliert wird.
- Wie Interrupts erkannt und kontrolliert werden.

10.1 Videoprioritäten

Die Prioritäten der Sprites untereinander können wir nicht ändern. Die Sprites mit der niedrigeren Nummer erscheinen immer vor den Sprites mit der höheren DMA-Kanal-Nummer. Diese Prioritäten sind durch die Arbeitsweise der Sprite-DMA-Kanäle bedingt. Sprites werden von der Hardware in vier Gruppen zu jeweils zwei Sprites behandelt. Auch diese Gruppierung ergibt sich aus der Arbeitsweise der Spritelogik. Die Gruppen sind:

Sprite 0 und 1

Sprite 2 und 3

Sprite 4 und 5

Sprite 6 und 7

Das Konzept der Videoprioritäten ist einfach zu verstehen, wenn man sich vorstellt, daß vier Finger der einen Hand die Spritepaare

repräsentieren und zwei Finger der anderen Hand die zwei möglichen Playfields repräsentieren. Genauso, wie wir nicht die Reihenfolge der Finger an unseren Händen ändern können, können wir auch nicht die relative Priorität der Sprites ändern. Wir können allerdings die zwei Finger der einen Hand zwischen die vier Finger der anderen Hand auf verschiedene Arten legen. Genauso können wir auch die Playfields vor oder hinter die Sprites legen. Insgesamt gibt es fünf mögliche Positionen für die Playfield-»Finger«. Eine andere Variation erreichen wir einfach dadurch, indem wir die zwei Playfield-Finger eins nach unten schieben. Jetzt sind Sprites 0, 1, 2 und 3 vor dem ersten Playfield, Sprite 4 und 5 erscheinen zwischen den beiden Playfields, und Sprite 6 und 7 erscheinen ganz hinten.

Die Zahlen 0 bis 4, die die Positionen der Playfield-Finger darstellen, sind die Werte, die wir in das Prioritäts-Kontrollregister schreiben, je nach gewünschter Priorität. Zusätzlich können wir die Priorität der beiden Playfields zueinander ändern, also quasi die beiden Finger vertauschen.

Diese Änderungen ermöglicht uns das Prioritäts-Kontrollregister, daß die Einstellung der Playfield-Prioritäten relativ zu Sprites und untereinander kontrolliert. Im Normalfall erscheint Playfield 1 vor Playfield 2. Das PF2PRI-Bit im Prioritäts-Kontrollregister dreht das um, so daß Playfield 2 vorne erscheint. Die Bits 5 bis 3 (PF2P2 – PF2P0) kontrollieren, wo zwischen den Spritegruppen Playfield 2 liegt, und die Bits 2 bis 0 (PF1P2 - PF1P0) kontrollieren die Position von Playfield 1.

10.2 Kollisionen

Wir können die Kontrollhardware auch dazu verwenden, Kollisionen zwischen einer Spritegruppe und einer anderen Spritegruppe oder zwischen einer Spritegruppe und einem der Playfields oder zwischen den zwei Playfields und so weiter zu erkennen.

Die erste Art der Kollision ist typisch für ein Computerspiel, bei dem es darum geht, feindliche Aliens abzuschießen. Hat die Rakete das Alien getroffen, müssen wir schließlich etwas unternehmen.

Die zweite Kollisionsart wird typischerweise dazu verwendet, ein sich bewegendes Objekt innerhalb der Bildschirmgrenzen zu halten.

Die dritte Kollisionsart schließlich erlaubt uns, Teile von Playfields als individuelle Objekte (BOBs) zu bewegen; das nennt man auch »Playfield Animation«. Wenn ein Playfield als Hintergrund (genannt »Playing Area«) benutzt wird, und das andere verwendet wird, um die Objekte zu definieren (BOBs), kann die Hardware Kollisionen zwischen diesen Playfield-Objekten und den Sprites wie auch Kollisionen zwischen den Playfield-Objekten und einem anderen Playfield erkennen.

Was ist eine Kollision?

Die Videoausgabe erfolgt, wenn alle Daten der Bitplanes und Sprites in einen gemeinsamen Datenstrom für das Display in die Videoausgabelogik gehen. Für jede Pixelposition auf dem Bildschirm wird die Farbe des Objektes mit der höchsten Priorität an den Bildschirm gesendet. Kollisionen werden erkannt, wenn zwei oder mehr Objekte gleichzeitig auf der selben Pixelposition sind. Dadurch wird ein Bit im Kollisionsdatenregister gesetzt. Eine Kollision ist also entstanden, wenn mehrere Bits des Ausgabedatenstroms auf dieselbe Bildschirmposition gerichtet sind.

Wie wir die Kollisionsdaten interpretieren

Das Kollisionsdatenregister, CLXDAT, kann nur gelesen werden (nicht geschrieben). Sein Inhalt wird automatisch auf 0 zurückgesetzt, nachdem er gelesen wurde. Auf der folgenden Seite finden Sie einen tabellarischen Überblick der Bits von CLXDAT.

Bitnummer	Registrierte Kollisionen
15	nicht benutzt
14	Sprite 4 (oder 5) mit Sprite 6 (oder 7)
13	Sprite 2 (oder 3) mit Sprite 6 (oder 7)
12	Sprite 2 (oder 3) mit Sprite 4 (oder 5)
11	Sprite 0 (oder 1) mit Sprite 6 (oder 7)
10	Sprite 0 (oder 1) mit Sprite 4 (oder 5)
09	Sprite 0 (oder 1) mit Sprite 2 (oder 3)
08	Gerade Bitplanes mit Sprite 6 (oder 7)
07	Gerade Bitplanes mit Sprite 4 (oder 5)
06	Gerade Bitplanes mit Sprite 2 (oder 3)
05	Gerade Bitplanes mit Sprite 0 (oder 1)
04	Ungerade Bitplanes mit Sprite 6 (oder 7)
03	Ungerade Bitplanes mit Sprite 4 (oder 5)
02	Ungerade Bitplanes mit Sprite 2 (oder 3)
01	Ungerade Bitplanes mit Sprite 0 (oder 1)
00	Gerade Bitplanes mit ungeraden Bitplanes

Tabelle 10.1: Die Bits von CLXDAT

Die Kontrolle der Kollisionserkennung

Das Kollisionskontrollregister, CLXCON, enthält die Bits, die einige Charakteristiken der Kollisionserkennung definieren. In dieses Register kann nur geschrieben werden. Lesen ist unmöglich.

Bitnummer	Name	Funktion
15	ENSP7	Enable Sprite 7
14	ENSP5	Enable Sprite 5
13	ENSP3	Enable Sprite 3
12	ENSP1	Enable Sprite 1
11	ENBP6	Enable Bit Plane 6
10	ENBP5	Enable Bit Plane 5
09	ENBP4	Enable Bit Plane 4
08	ENBP3	Enable Bit Plane 3
07	ENBP2	Enable Bit Plane 2
06	ENBP1	Enable Bit Plane 1
05	MVBP6	Match Value Bitplane 6 Collision
04	MVBP5	Match Value Bitplane 5 Collision
03	MVBP4	Match Value Bitplane 4 Collision
02	MVBP3	Match Value Bitplane 3 Collision
01	MVBP2	Match Value Bitplane 2 Collision
00	MVBP1	Match Value Bitplane 1 Collision

Tabelle 10.2: Die Bits von CLXCON

Die Bits 15 bis 12 definieren, daß Kollisionen mit einem Spritepaar auch bei ungeraden Spritenummern erkannt werden sollen. Die geradzahligen Sprites werden immer berücksichtigt, die ungeraden nur, wenn sie durch Setzen dieser Bits für die Kollision »angeschaltet« sind.

Die Bits 11 bis 6 definieren, ob bestimmte Bitplanes des Playfields aus der Kollisionserkennung ausgeschlossen beziehungsweise in die Erkennung eingeschlossen werden sollen.

Die Bits 5 bis 0 definieren die Polarität (TRUE/FALSE-Bedingung) von Bits, die eine Kollision verursachen. Wir wollen zum Beispiel, daß die Hardware die Kollision nur erkennt, wenn das Objekt mit etwas Grünem oder etwas Blauem kollidiert. Diese Fähigkeit zusammen mit den Kollisions-Enable-Bits (15 - 12) erlaubt uns, die

exakten Bits und ihre Polarität für die zu registrierenden Kollisionen festzulegen.

10.3 Die Rasterstrahlzähler

Insgesamt gibt es vier Adressen, die auf den Rasterstrahlzähler zugreifen können. Dieselbe Adresse wird allerdings auch benutzt, um die Lightpen-Position zu lesen anstatt die Rasterstrahlposition, sofern wir einen Lightpen (Lichtgriffel) verwenden.

Das Register VPOSR kann nur gelesen werden. Bit 15 enthält das High Bit der vertikalen Position (V8) und Bit 0 das »Frame Type Bit« (Interlace/Non-Interlace). Die Bits 14-1 sind unbenutzt.

Das Register VHPOSR kann ebenfalls nur gelesen werden. Die Bits 15-8 enthalten die Low Bits der vertikalen Position, V7 bis V0. Die Bits 7-0 (H8 bis H1) enthalten die horizontale Position.

Das Register VPOSW kann nur beschrieben werden und enthält dieselben Bits wie VPOSR. Entsprechend enthält das beschreibbare Register VHPOSW die gleichen Bits wie VHPOSR.

10.4 Interrupts

Das Custom-Chip-System unterstützt alle Arten des 68000-Prozessorinterrupts. Die verschiedenen Arten von Interrupts werden von den einzelnen Hardware-Einheiten an den Peripheriechip Paula gesendet, der sie in 6 der 7 möglichen Interrupts des 68000 umwandelt. Der Paula-Chip erzeugt die 68000-Interrupt-Levels 1 bis 6 (maskierbare Interrupts). Die Kontrollregister innerhalb des Paula-Chips erlauben uns, verschiedene dieser Quellen zu »maskieren« und sie damit davon abzuhalten, eine Unterbrechung des 68000 zu erzeugen. Die Systemsoftware des Amiga wurde dazu entwickelt, alle Hardwareinterrupts von Level 1 bis 6 korrekt zu verwalten. Insgesamt gibt es zwei Interrupt-Register, Interrupt Enable (Maske) und Interrupt Request (Status). Jedes der Register hat sowohl eine Lese- als auch eine Schreib-Adresse.

INTENA (Interrupt Enable Mask) kann nur beschrieben werden; setzt oder löscht bestimmte Bits des Interrupts.

INTENAR (Interrupt Enable Read) kann nur gelesen werden und enthält die Bits aus INTENA.

INTREQ (Interrupt Request) kann nur beschrieben werden und wird vom Prozessor benutzt, um eine bestimmte Art von Interrupt (Software Interrupt) durchzuführen. Es wird außerdem benutzt, um die Interrupt-Request-Flags zu löschen, sobald der Interrupt-Prozeß durchgeführt ist.

INTREQR (Interrupt Request Read) kann nur gelesen werden und enthält die Bits, die definieren, welche Dinge eine Interrupt-Unterstützung benötigen.

Die Bitpositionen im Interrupt-Request-Register entsprechen direkt den Bits der selben Positionen im Interrupt-Enable-Register. Der einzige Unterschied zwischen den »Nur-Lese«- und »Nur-Schreib«-Registern ist, daß Bit 15 in der Nur-Lese-Adresse keine Bedeutung hat.

Setzen und Löschen der Interrupt-Bits

Sowohl die Interrupt-Register als auch die DMA-Kontrollregister benutzen eine spezielle Art, ihre Bits zu setzen oder zu löschen. Bit 15 dieser Register wird das SETCLR-Bit genannt. Wollen wir ein Bit setzen (eine 1 hineinsetzen), müssen wir zuerst eine 1 in die Position setzen, die wir auf 1 haben wollen, und anschließend eine 1 in Position 15 schreiben. Wenn wir ein Bit löschen wollen (eine Null hineinsetzen), müssen wir zuerst eine 1 dort hineinsetzen, wo wir die 0 haben wollen, und anschließend eine 0 in Position 15 schreiben.

Die Positionen 14 bis 0 sind sogenannte »Bit-Selektoren«. Wir schreiben Einsen in die Bits, die wir »selektieren«, also anwählen wollen. Zur selben Zeit schreiben wir entweder eine 0 oder eine 1 in Bit 15, um die angewählten Bits entweder zu setzen oder zu löschen. Diese Bits haben folgende Funktionen:

Bit 14, INTEN, ist das sogenannte »Master Interrupt Enable Bit«. Wenn dieses Bit auf 0 gesetzt wird, werden alle anderen Interrupts abgeschaltet. Das Löschen dieses Bits können wir beispielsweise dazu verwenden, alle Interrupts abzuschalten, um einen zeitkritischen Einsatz des 68000 zuerst zu Ende zu bringen. Dieses Bit erzeugt keinen »Interrupt Request« (»Verlangen« einer Unterbrechung vom Prozessor), sondern wird lediglich für das An- und Abschalten verwendet.

Die Bits 13 (EXTER) und 3 (PORTS) des Interrupt-Registers sind reserviert für die Erzeugung von Interrupts durch externe Peripherie (über den Erweiterungsbus).

Bit 5, VERTB, verursacht eine Unterbrechung bei Zeile 0 (Start des vertikalen Blankings) des Videodisplays. Es wird oft vom System verlangt, viele unterschiedliche Dinge zu tun, während das vertikale Blanking erfolgt. Zu diesen Aufgaben gehört das Zurücksetzen der verschiedenen Adreßzeiger, das Neuschreiben von Copperlisten und andere Systemkontroll-Operationen. Da wir nicht alles mit dem Copper machen können (zum Beispiel die Berechnung der Bitplanezeiger für diagonales Scrolling des Playfields), muß beim vertikalen Blanking der Prozessor eingesetzt werden, um diese Aufgaben zu erledigen. Dazu wird bei jedem vertikalen Blanking dieses Interrupt-Signal über Paula an den Prozessor gesendet.

Bit 4, COPER, wird vom Copper verwendet, um ein Interrupt-Signal zu senden. Im Normalfall wird dieses Bit benutzt, wenn wir erfahren wollen, ob der Rasterstrahl eine bestimmte Position erreicht hat und dann etwas im Speicher aufgrund dieses Vorfalls ändern.

Die Bits 10-7 sind den Audio-Kanälen zugeordnet. Da wir uns hier nicht mit Musik beschäftigen, werden diese Interrupts hier nicht beschrieben. Hier wird ein Level-4-Interrupt erzeugt.

Bit 6, BLIT, signalisiert, daß der Blitter seine Arbeit beendet hat. Wenn dieses Bit auf 1 gesetzt ist, ist die Blitter-Arbeit beendet. Der Blitter ist jetzt bereit, weitere Aufgaben zu erledigen. Wir verwenden diesen Interrupt dazu, erst dann auf den Blitter zuzugreifen (seine Register für neue Aufgaben zu setzen), wenn er mit der letzten Arbeit fertig ist.

Die Bits 12 und 1, DSKSYN und DSKBLK, sind Bits zur Kontrolle der Diskettenlaufwerke.

Bit 11, RBF, und Bit 1, TBE sind Interrupts für die Kontrolle des seriellen Ports und seiner Buffer.

Buchteil 3

Dieser Teil des Buches ist all denen gewidmet, die mit AmigaBASIC in die Welt der Computergrafik einsteigen wollen. Auch dieser Teil kann unabhängig von den anderen Buchteilen gelesen werden. Nach der Einführung in die BASIC-Befehle und ihren Anwendungen für Grafik greifen wir allerdings ein kleines bißchen auf Grundwissen des ersten Buchteils zurück – dazu genügt aber ein kurzes Nachschlagen in den entsprechenden Kapiteln, vollständig lesen brauchen Sie den ersten Buchteil dazu nicht unbedingt.



11

Einleitung

BASIC ist schon seit sehr langer Zeit die Sprache, die einfach serienmäßig zu einem Computer gehört. Damit kann auch der Anfänger in einer einfach erlernbaren Sprache mit seinem Computer mehr tun, als nur fertige professionelle Software zu benutzen. BASIC kann man fast als Standard bezeichnen – doch mit Standard habe ich gleich wieder den wunden Punkt getroffen. Denn es gibt keinen echten Standard für BASIC. BASIC gibt es in Dutzenden von verschiedenen Dialekten – jeder BASIC-Interpreter oder -Compiler hat seine Vor- und Nachteile.

11.1 BASIC oder nicht BASIC – das ist die Frage

Wenn wir uns einmal die grafischen Fähigkeiten des Amiga vor Augen führen, ist eigentlich schon klar, warum es keinen Standard geben kann: Es muß ein BASIC existieren, das die Fähigkeiten des Computers auch ausnutzt, anstatt sich an ein starres Schema zu halten. Die ach so tollen Fähigkeiten eines neuen Computers hätten kaum einen Zweck, wenn wir sie nicht von eigenen Programmen aus erreichen können.

Für den Amiga gibt es bereits mehrere BASIC-Versionen: Amiga-BASIC von Microsoft, ABASIC von Metacomco und TrueBASIC von TrueBASIC Inc. Jede dieser Versionen hat Vor- und Nachteile. ABASIC nutzt die Grafikfähigkeiten des Amiga am besten aus, ist dafür aber das bedienerunfreundlichste BASIC, das mir seit Ende der siebziger Jahre untergekommen ist. TrueBASIC ist das bedienerfreundlichste BASIC, das ich kenne (Editor mit Mausbedienung und

allem Komfort), leidet aber an der Politik der Herstellerfirma, für alle Personalcomputer identische Grafikbefehle zu verwenden. Das bedeutet: Mit diesem BASIC können wir aus dem Amiga kaum mehr herausholen, als aus einem MS-DOS-Rechner. Das AmigaBASIC von Microsoft schließlich ist furchtbar langsam im Vergleich zu den anderen Versionen, bietet aber fast den idealen Kompromiß zwischen Bedienerfreundlichkeit und Ausnutzung der Hardware.

Könnte ich auswählen, mit welchem BASIC ich arbeiten möchte, fiele meine Wahl auf ein Hybrid-BASIC aus allen Dreien: Die Grafikfähigkeiten und Geschwindigkeit von ABASIC, die strukturierte übersichtliche Programmierung des AmigaBASIC ohne Zeilennummern, der Editor von TrueBASIC, und fertig ist das Wunderkind. Leider können wir aber nicht wählen. Das soll heißen: Microsoft-BASIC wird mit dem Amiga ausgeliefert und wir müssen zunächst damit auskommen. Die geringe Geschwindigkeit eignet sich nicht gerade für schnelle Grafik und Animation – doch Sie werden sehen, daß man damit trotzdem Unglaubliches aus der kleinen Wunderkiste Amiga zaubern kann, was in erster Linie der Amiga-Hardware zu verdanken ist.

11.2 Was Sie erwartet

Die nachfolgenden Kapitel sollen Sie mit den Grafikbefehlen des AmigaBASIC vertraut machen, um anschließend in Übungen und Beispielen das Wissen vertiefen zu können. Nach verschiedenen Möglichkeiten, statische (unbewegte) Grafik auf dem Amiga darzustellen und zur grafischen Unterstützung eigener Programme zu verwenden, werden Sie in die grundlegenden Möglichkeiten der Animationsprogrammierung in BASIC eingeführt. Neben der Erläuterung verschiedener Techniken finden Sie auch Beispiele, die Ihnen das Verständnis erleichtern sollten.

Ich setze allerdings voraus, daß gewisse Grundkenntnisse in BASIC vorhanden sind. Wer eine etwas langsame und behutsam vorgehende Einführung in die Sprache haben will, dem kann ich nur wärmstens das Buch «Amiga-Programmier-Praxis mit MSBASIC« von David A. Lien, erschienen im tewi-Verlag, empfehlen. Das Buch bietet eine detaillierte Einführung für Absolut-Anfänger und ist zudem unterhaltsam geschrieben.

12

Die Grafikbefehle des AmigaBASIC

12.1 Grundlegende geometrische Objekte

Wie bereits im ersten Buchteil erklärt, besteht eine Grafik grundsätzlich aus einigen wenigen geometrischen Objekten: Punkte, Linien, Kreise, Ellipsen, Rechtecke und so weiter. Sie erfuhren auch, wie die geometrischen Objekte grundsätzlich auf Computern dargestellt werden (Umsetzung Geometrie zu Pixelgrafik). Dieses Wissen ist zum Arbeiten mit BASIC aber nicht unbedingt nötig. Die Umsetzung der Geometrie auf Pixelgrafik, das Setzen oder Löschen einzelner Bits, übernimmt der BASIC-Interpreter für uns. Wir müssen uns nur noch um die Start- und Endkoordinaten, eventuell auch um einen Radius oder andere Hilfsparameter kümmern.

12.1.1 Koordinaten und andere Grundlagen

Wollen wir eine Linie von links oben nach rechts unten ziehen, geben wir lediglich die x/y-Koordinaten des linken oberen Punktes und des rechten unteren Punktes an. Die Pixels dazwischen, die unsere Linie bilden, setzt der BASIC-Interpreter für uns.

Um Koordinaten angeben zu können, benötigen wir jedoch zunächst ein Koordinatensystem. Das Koordinatensystem des AmigaBASIC beginnt mit den Koordinaten (0,0) in der linken oberen Ecke (nicht wie im Mathematikunterricht links unten). Die Endkoordinaten sind bestimmt durch die gewählte Grafikauflösung (siehe Screen-Befehl) und die Windowgröße (siehe Window-Befehle).

Wir müssen also, wollen wir eine mathematische Funktion auch so wie im Mathematikunterricht darstellen, die Koordinaten umrechnen. Dazu genügt die Umrechnung des y-Wertes, da die x-Werte nach wie vor von links nach rechts gezählt werden, der y-Wert aber »verkehrt herum« steht. Eine Umrechnung bewerkstelligen wir mit folgender Formel:

ye = ya - y

wobei ye = »y endgültig«, ya = »y-Auflösung«, y = ursprünglicher y-Wert.

Sollen also mathematische Funktionen dargestellt werden, rechnen wir vor oder während der Ausführung des Grafikbefehls die y-Koordinate auf diese Weise um.

12.1.2 Punktbefehle

Das kleinste Element einer Grafik ist ein Pixel. Zum Setzen, Löschen oder Ändern eines solchen Bildschirmpunktes stehen uns die Punktbefehle zur Verfügung.

Der Befehl zum Setzen eines Punktes lautet:

PSET 'STEP' (x,y) ',Farbe'

Die in »'« stehenden Teile der Befehlssyntax sind nicht notwendig, sondern lediglich zusätzliche Optionen. Es würde also ein reines PSET (x,y) genügen, um einen Punkt zu setzen. Der Punkt wird dabei in derjenigen Farbe gezeichnet, die durch den COLOR-Befehl (siehe Kapitel »Farben«) als Vordergrundfarbe deklariert wurde. Vordergrundfarbe ist die Farbe, mit der auch Schrift in PRINT-Befehlen ausgegeben wird.

Wird eine Farbe als Zahl von 0 bis 31 angegeben, so erscheint der Punkt in der gewünschten Farbe. Die Anzahl der wirklich verfügbaren Farben ist abhängig von der mit dem Screen-Befehl gewählten Bildschirmtiefe (Tiefe = Anzahl der Bitplanes für einen Grafikbildschirm. Vergleichen Sie hierzu auch den Hardwareteil dieses Buches.). Wird eine Zahl angegeben, die höher ist als die Anzahl der verwendbaren Farben, reagiert der Amiga mit einer Fehlermeldung.

Der Zusatz STEP wird auch »Offset-Angabe« genannt. Das heißt, der Punkt wird um einen »Offset«, eine Abweichung vom zuletzt gemalten Punkt von x Pixels nach rechts und y Pixels nach unten dargestellt. Haben wir also erst einen Punkt bei (160,80) gesetzt, wird der nächste Punkt bei *PSET STEP (10,10)* an der Koordinate (170,90) dargestellt. Soll der Offset nach oben statt nach unten, oder soll er nach links statt nach rechts gehen, können auch negative Werte für x und y angegeben werden. Wichtig ist nur, daß das Gesamtergebnis aus ursprünglicher Position des Grafikcursors und Offset nicht negativ wird, denn auf negative Koordinaten kann der Amiga nicht zeichnen.

Löschen können wir einen Punkt, indem wir einfach wieder den PSET-Befehl aufrufen, diesmal aber mit der Hintergrundfarbe als Farbangabe. Wird im Programm viel mit Farben hantiert, kann es aber schon mal passieren, daß dem Programmierer die Hintergrundfarbe nicht mehr ganz klar ist. Damit wir uns nicht im Farbendilemma verstricken, haben uns die Macher des AmigaBASIC aber glücklicherweise einen Alternativbefehl beschert.

PRESET 'STEP' (x,y) ',Farbe'

Die Syntax und die Optionen sind voll identisch zum PSET-Befehl. Einziger Unterschied zu PSET ist: Geben wir keine Farbe an, wird der Punkt auf die Hintergrundfarbe gesetzt und damit gelöscht.

Da es für Anfänger, und besonders des Englischen nicht Kundige, nicht immer so leicht ist, sich Befehle zu merken, werde ich zu jedem BASIC-Befehl schreiben, was die Kürzel eigentlich bedeuten: PSET steht für »Pixel-Set«, also »Pixel Setzen«. »PRESET« bedeutet »Pixel Reset«, also »Pixel Zurücksetzen« auf Hintergrundfarbe. Ich hoffe, diese kleinen Eselsbrücken helfen Ihnen.

Manchmal kann es wichtig sein, die Farbe eines Punktes auf dem Bildschirm zu wissen. Beispiel: Wir sind gerade dabei, ein Computerspiel zu programmieren. Hat bei Programmablauf unsere Kollisionsabfrage festgestellt, daß unser UFO auf eine Hintergrundgrafik gestoßen ist, müssen wir natürlich wissen, ob es ein Fels ist oder ob unser UFO nur über den Sternenhintergrund fliegt. Zu diesem Zweck sind die Sterne als hellblaue Punkte gemalt, der Fels in dunklem Braun. Hellblau haben wir als Farbe 2 deklariert, dunkelbraun als Farbe 3 (siehe Farbbefehl PALETTE). Die Position des UFOs ist uns

bekannt. Den Punkt an dieser Position fragen wir nun ab. Dies leistet die folgende Funktion.

POINT (x,y)

Diesmal haben wir es nicht mit einem Befehl, sondern mit einer Funktion zu tun. Funktionen werden im allgemeinen Variablen zugewiesen, also beispielsweise *a=POINT(x,y)*, oder werden direkt von einem PRINT-Befehl ausgegeben: *PRINT POINT (30,40)*.

Doch zurück zu unserem Beispiel eines Computerspiels: Natürlich müssen wir nach Abfrage des Farbwertes durch den POINT-Befehl entsprechend mit IF-Befehlen auf Farbe 2 oder 3 reagieren. Das könnte dann beispielweise so vor sich gehen, wie in dem folgenden Listing.

```
Abfrage:
 a = POINT (x,y)         'Farbabfrage
 IF a = 2 THEN Ablauf    'nur Sterne
 IF a = 3 THEN GOSUB Boom 'Fels, also Explosion
    .
    . 'weitere Abfragen
    .
Ablauf:
    .
    . 'Programm geht weiter
    .
 END 'Ende Hauptprogramm
Boom: 'Unterprogramm
    .
    . 'Hier Explosionsroutine
    .
RETURN
```

Allein mit den Punktbefehlen sind wir nun schon in der Lage, unser erstes kleines Grafikdemo zu schreiben. Wie das Dots-Demo auf der Ihnen wohl bekannten »Workbench Demos«-Diskette schreiben wir unser eigenes kleines Punktedemo:

```
WINDOW 1   'Basic-Outputwindow für Grafik verwenden
FOR i=1 TO 1000     '1000 Punkte malen
x=RND(1)*630        'Zufallswert x-Koordinate
y=RND(1)*200        'Zufallswert y-Koordinate
PSET (x,y),RND(1)*3 'Punkt mit Zufallsfarbe 0 bis 3
NEXT I
```

Zugegeben, es sieht nicht gerade überwältigend aus, ist aber doch schon mal ein erstes Erfolgserlebnis. Nachdem wir nicht eine ganze Grafik nur mit dem Setzen von Punkten erstellen wollen (was viel zu lange dauern würde), wenden wir uns nun den Linien zu.

12.1.3 Linien, Rechtecke und Muster

Linien sind die Grundlagen der animierten 3D-Grafik, wie Sie sicher schon an den größeren Grafiksystemen gesehen haben. Dreidimensionale Objekte werden meist als Gitternetz aus einzelnen Linien entworfen. Erst später kommt dann das Ausfüllen der Zwischenflächen und das Glätten der Ecken und Kanten. Wir wollen uns zu Beginn unserer Exkursionen aber lieber noch in zwei Dimensionen aufhalten. Zum Zeichnen einer Linie rufen wir LINE- Befehl auf.

LINE 'STEP'(x,y)-'STEP'(x,y)',Farbnummer",b'f"

Viele der Optionen benötigen wir nicht, so zum Beispiel die Offsetangaben mit STEP, die Farbnummer, oder die Optionen b und f, auf die ich später noch näher eingehen werde. Eine Linie von links oben nach rechts unten malt das folgende Beispiel auf den Bildschirm.

```
LINE (10,10)-(100,100)
```

Das genügt schon, um eine saubere Linie zu malen. Erinnern Sie sich, wie die Software intern eine Linie berechnet? (Grundlagenteil). Zum Geschwindigkeitsvergleich hier eine kurze Routine in BASIC, mit der wir eine Linie zeichnen würden, wenn wir keinen LINE-Befehl hätten. Das nachfolgende Listing entspricht dem Programmablaufplan in Abbildung 1.8 aus dem ersten Buchteil.

```
REM Line-Algorithmus
start:
 INPUT "Koordinaten x1,y1,x2,y2 ",x1,y1,x2,y2
 CALL Linie (x1,y1,x2,y2)
ende:
 END

'Unterroutine Line-Algorithmus:
SUB Linie (x1,y1,x2,y2) STATIC
 'x1,y1 = Startkoordinaten
 'x2,y2 = Endpunkt
 'dx,dy = zu zeichnender Vektor
 'steps = Anzahl der Schritte am Vektor
 'xs,ys = Größe eines Steps
 'x,y   = Punktekoordinaten
 'u     = Zählvariable für Schritte
 'z     = 1 wenn Linie rechts, -2 bei Linkslinien

begin:
 dx = x2-x1 : dy = y2-y1      'Vektor bestimmen
 IF dx>dy THEN steps=dx ELSE steps=dy
 '(Anzahl der Schritte bestimmen)
 xs = dx/steps 'Schrittgroesse bestimmen
 ys = dy/steps
 x = x1+.5  'Grundposition für Rundungen
 y = y1+.5
 z=1 : IF x2 < x1 THEN z=-1
  '(Schrittweite 1 nach rechts gehen = z=1)
  '(Zielpunkt links vom Anfang =  z=-1)
 FOR u=1 TO steps STEP z
  PSET (INT(x),INT(y))   'Punkt setzen
   x=x+xs : y=y+ys       'nächster Punkt
 NEXT
END SUB
```

Aus Linien zusammengesetzt werden auch weitere geometrische Elemente, so zum Beispiel Rechtecke. Wie Sie an der LINE-Routine sehen, ist das Grundelement einer Grafik der Punkt. Aus Punkten werden Linien, und aus Linien werden Rechtecke und Polygone. Zuerst geht es uns jedoch nur um die Rechtecke: Da sie nichts weiter sind als nur vier Linien, dachten sich die AmigaBASIC-Macher, sie doch gleich als Option in den LINE-Befehl einzubauen. Andere BASIC-

Dialekte benützen dazu extra einen Box-Befehl, doch wozu eine zusätzliche Routine verwenden, wenn wir die wichtigsten Elemente schon in der LINE-Routine haben? Gesagt, getan, und schon war die »b«-Option des LINE-Befehls geboren. Als Startkoordinate geben wir die linke obere Ecke an, als Endkoordinate die rechte untere Ecke.

LINE (x1,y1)-(x2,y2),Farbe,b

Dieser Befehl zeichnet nun keine Linie zwischen den beiden angegebenen Punkten, sondern ein Rechteck (dessen Diagonale die Linie wäre). Lassen wir die Farbe weg (um ganz einfach die Vordergrundfarbe zu verwenden), müssen wir (ganz wichtig!) zwei Kommas vor dem b tippen. Also beispielsweise *LINE (10,10)-(80,80),,b.*

Wollen wir die Fläche unseres Vierecks gefüllt haben, genügt ein weiterer Buchstabe, das »f«. Das b steht für »Box«, also Rechteck. Das »f« steht für »filled«, also »gefüllt«.

LINE (x1,y1)-(x2,y2),Farbe,bf

Das gleiche, was wir bereits mit unserem Punkte-Demo gemacht haben, läßt sich natürlich nun auch mit den Rechtecken machen: Aus dem PSET-Befehl wird dabei ein »LINE...,,bf«, und zwei weitere Zufallszahlen für die Größe des Rechtecks müssen eingeführt werden.

Linien, Linien und kein Ende. Denn außer dem Füllen der Rechtecke mit Farben können sie auch mit Mustern gefüllt werden. Das gleiche gilt für Linien: Statt mit einer Farbe können sie mit Mustern gezogen werden. Der dazu nötige Befehl lautet:

PATTERN (Linienmuster)(,Polygon-Muster)

Die zweite Muster-Option wird später für uns interessant, wenn wir zu den Polygonen schreiten. Die erste Option des Befehls legt das Muster fest, mit dem Linien und Kanten der Rechtecke gemalt werden. Der Wert, den wir als Muster dort eintragen, ist eine sogenannte 16-Bit-Maske. Eine Maske ist im Zusammenhang mit der Grafik immer die Zahlenrepräsentation einer Bitfolge.

Die Grafik, die in diesem Fall von der Maske repräsentiert wird, ist ein 16 Punkte breites Muster. Ist die Maske 1010101010101010, wobei 1

für gesetzter Punkt steht und 0 für nicht gesetzter Punkt, sieht eine mit diesem Muster ausgefüllte Box wie folgt aus:

1010101010101010
1010101010101010
1010101010101010
1010101010101010

Im PATTERN-Befehl geben wir dabei die dezimale Entsprechung dieser 0-1-Kombination an. Mit & vor der Zahl kann auch eine oktale Angabe und mit &H vor der Zahl auch eine hexadezimale Angabe erfolgen.

Mit dem bisher gelernten Stoff können wir bereits erste Grafik-Anwendungen schreiben. Nur mit Linien und Punkten ist beispielsweise ein Funktionsplotter machbar, mit Hilfe der Rechtecke lassen sich Balkendiagramme erstellen. Und haben wir viele Balken darzustellen, wollen aber nicht viel Speicherplatz für mehr Farben verschwenden, können wir die Balken auch durch verschiedene Muster (Pattern-Befehl) voneinander unterscheiden.

12.1.4 Polygone

Mit unseren bisherigen Kenntnissen stehen wir aber erst am Anfang unserer Grafikträume. Wir haben bisher nur die Anzahl der Enden mit 2 potenziert: Von einem »Ende« (Punkt) auf zwei Enden (Linie) bis schließlich zu vier Enden (Rechtecke). Kommt es zu ungeraden Zahlen, müssen wir unsere Objekte aus einzelnen Linien zusammensetzen. Zumindest, wenn wir sie nicht ausfüllen wollen. Aber da haben wir noch das Betriebssystem des Amiga, das eine Routine besitzt, die unter Ausnutzung des Blitters sehr schnell ausgefüllte Polygone zeichnen kann. Das AmigaBASIC nutzt diese Routine aus: Mit den AREA-Befehlen brauchen nur die Eckpunkte eines Polygons festgesetzt zu werden, das mit dem AREAFILL-Befehl schließlich sehr schnell ausgefüllt werden kann.

AREA 'STEP' (x,y)

AREA legt jeweils einen Eckpunkt des in Entstehung befindlichen Polygons fest. Die STEP-Option ermöglicht es uns, mehrere gleich große und gleich aussehende Polygone zu malen, ohne jeden Eckpunkt neu zu berechnen. Lediglich der Anfangspunkt muß festgesetzt werden. Damit haben wir ein mächtiges Instrument für schnelle zwei-

dimensionale Animation, mit der wir allerdings auch dreidimensionale Effekte erzielen können. Sind alle Eckpunkte festgelegt, wird das Polygon mit dem folgenden Befehl gefüllt.

AREAFILL

Hier gibt es leider keine Optionen. Farbangaben lassen die Polygon-Befehle leider nicht zu, gezeichnet wird immer in der durch den COLOR-Befehl gewählten Vordergrundfarbe. Vor dem Zeichnen eines Polygons muß also immer die gewünschte Farbe festgesetzt werden. Soll das Polygon mit einem Muster ausgefüllt werden, muß dies vorher mit dem PATTERN-Befehl festgelegt werden. Für die sogenannte »Area-Maske« gilt das gleiche wie für die Line-Maske. Soll keine Line-Maske erstellt werden, gilt auch hier wieder die »2 Komma-Regel«: *PATTERN,,16-Bit-Maske* setzt nur die Polygon-Maske fest.

Zum besseren Verständnis folgt wieder die Übersetzung der Befehle: AREA heißt auf deutsch »Bereich«. Mit diesem Befehl setzen wir also einen Grafikbereich fest, der gefüllt werden soll. AREAFILL bedeutet »Füllen des Bereichs«; dadurch wird der festgelegte Polygonbereich gefüllt und somit auf den Bildschirm gezeichnet.

Die Möglichkeiten der Polygonbefehle liegen eindeutig in der schnellen Animation. Ein kommerzielles Programm, das im wesentlichen dieselbe Routine ausnützt wie der AREA- und AREAFILL-Befehl, ist der »Animator« von Aegis (vergleiche das Kapitel »kommerzielle Grafiksoftware«). Der Animator erzielt durch die Nutzung der Polygon-Routinen eine extrem hohe Geschwindigkeit und kann daher Realzeit-Animation durchführen (das heißt, die Berechnung der Bewegung erfolgt während der Grafikdarstellung und nicht vorher). Ob eine Realzeitanimation in BASIC möglich ist, hängt von den Berechnungsalgorithmen und dem Zweck der Animation ab. Grundsätzlich sollte man in BASIC aber alle Bewegungen vorher berechnen und erst anschließend ausführen. Näheres dazu in einem folgenden Kapitel, in dem wir verschiedene Animationstechniken besprechen werden.

Als Beispiel für die Geschwindigkeit der Polygonbefehle folgt nun ein kleines BASIC-Programm. Das Programm tut nichts weiter, als an der Stelle, an der der Mausbutton gedrückt wurde, ein Dreieck zu zeichnen. Durch die Abänderung der Werte *x1* bis *y2* läßt sich natürlich auch ein größeres oder kleineres Dreieck erzeugen. Durch Ver-

mehrung der Werte und der Polygonbefehle kann man so auch einen Stern oder andere Objekte zeichnen. Durch Festsetzen eines Musters mit dem Pattern-Befehl kann man so auch mit einem Muster malen. Die Geschwindigkeit ist ausreichend, um das Polygon auch als Pinsel zum Malen zu verwenden, wie sie an der Ausführung des Programmes sehen werden.

```
REM Polygondemo
 x1 = 10 : y1 = 0  'Dreieck definieren durch Offsets
 x2 =-10 : y2 =-10 'von x/y-Startposition
REM Hauptprogramm
 MOUSE ON

 Abfrage:
  IF MOUSE(0) <>0 THEN Malen: 'Mausbutton gedrückt, malen
  GOTO Abfrage               'Wenn nicht, dann warten.

 Malen:
  AREA (MOUSE(1),MOUSE(2))    'Ersten Punkt des Polygons
                              'x/y-Position der Maus
  AREA STEP (x1,y1)
  AREA STEP (x2,y2)
  AREAFILL                    'Polygon malen
  GOTO Abfrage                'zurück
```

12.1.5 Ellipsen und Kreise

Eine runde Sache braucht ein Programmierer, wenn er ein Erfolgserlebnis haben will. Deshalb weg von den Ecken, hin zu den Kreisen! Das AmigaBASIC bietet uns dafür den CIRCLE-Befehl. CIRCLE heißt Kreis, und genau den zeichnen wir mit dem folgenden Befehl:

CIRCLE 'STEP' (x,y),Radius ',Farbe,Start,Ende,Aspect'

Die einfachste Methode, einen Kreis zu zeichnen, ist *CIRCLE (x,y),Radius*. So wird der Kreis um den Punkt (x,y) herum gemalt (das heißt (x,y) ist der Mittelpunkt). Der Radius ist NICHT gleich der Zahl der Pixels, die das weiteste Ende des Kreises vom Mittelpunkt entfernt ist. Vielmehr hängt er vom Verhältnis der x- und y–Pixels des Bildschirms zueinander ab. Beim Standard-BASIC-Bildschirm sind das 640 Punkte horizontal zu 200 Punkten vertikal, was ein Verhältnis von 2,25 ergibt. Das heißt, daß ein Kreis, damit

er rund ist, in der Breite einen größeren Radius als in der Höhe haben muß. Der Radius gibt den Wert an, wieviele Pixels weit es bei einem 1:1-Verhältnis der horizontalen und vertikalen Auflösung vom Mittelpunkt bis zum Rand des Kreises wäre.

Die Farbangabe ist optional, bei Nichtangabe wird in der gewählten Vordergrundfarbe gezeichnet. Zu den Start- und Endwerten kommen wir später, denn sie stellen etwas dar, das bisher von noch fast keinem BASIC-Dialekt geboten wurde. Der Wert »Aspect« ist ein Streckungsfaktor. Verschiedene Monitore können nämlich ein verzogenes Bild haben, weswegen ein mit dem CIRCLE-Befehl gezeichneter Kreis nicht immer ganz rund, sondern eiförmig erscheint. Um nicht am Monitor herumdrehen zu müssen, kann man den Kreis deshalb mit dem Aspect-Wert dehnen und quetschen. Zumindest ist dieser Befehl dafür vorgesehen. Daß man dadurch natürlich auch Ellipsen verschiedener Größe darstellen kann, ist eigentlich die logische Konsequenz des Streckens und Quetschens. Der Standardwert für einen ganz runden Kreis sollte bei 0,44 liegen (abhängig von der Bildschirmauflösung). Wird kein Aspect-Wert angegeben, wird automatisch dieser Wert angenommen. Wie das Ergebnis bei einer Änderung aussieht, zeigt das folgende kleine Demoprogramm. (Bitte nicht vergessen: Genau vier Kommas vor dem Aspect-Wert!)

```
Aspect=.1
WHILE Aspect < 20
 CIRCLE(60,60),55,,,,Aspect
 Aspect = Aspect*1.4
WEND
```

Wie Sie sehen, ist das ein Auszug aus dem Demo »Picture«, das auf Ihrer BASIC-Diskette in der BASICdemos-Schublade abgelegt ist. Mehrere Ellipsen zusammen können natürlich ein schönes Bild abgeben, wie man hier auch gut sieht.

Die Angaben »Start« und »Ende« im CIRCLE-Befehl geben uns die Möglichkeit, einen Kreis nur teilweise zu zeichnen – also einen Kreissektor auf den Bildschirm zu bringen. Das erweist sich natürlich ideal für die sogenannten »Pie-Charts« oder Kuchengrafiken. Start gibt den Startwinkel an (im Bogenmaß), Ende gibt den Winkel an, bei dem der Teilkreis enden soll. Die Werte für Start und Ende müssen in Bogenmaß angegeben werden und können zwischen -2*pi und 2*pi liegen. Da wir normalerweise nicht im Bogenmaß rechnen,

sondern im Winkelmaß, sollte ich wohl am besten eine Erläuterung und eine kleine Umrechnung bringen.

Hier ist sie: Unser Winkelmaß geht von 0 bis 360 Grad. Eine Umrechnung vom Winkelmaß auf Bogenmaß geschieht durch eine Multiplikation mit 0.0174444444444444.... Der Einfachkeit halber begnügen wir uns mit 0.017444. Das AmigaBASIC malt seine Kreise gegen den Uhrzeigersinn, und beginnt am rechten äußeren Ende des Kreises. Das heißt, die Position 0 Grad ist nicht oben, sondern rechts auf gleicher Höhe mit dem Mittelpunkt. Die Winkel im AmigaBASIC werden gegen den Uhrzeigersinn gezählt. Wollen wir also einen Dreiviertelkreis malen, der oben beginnt und rechts aufhört, ist der Startwinkel 90 Grad,und der Endwinkel 360 Grad. Das sieht dann beispielsweise so aus:

```
CIRCLE(320,100),100,,90*0.017444,360*0.017444
```

Für eine schöne Kuchengrafik bräuchten wir jetzt nur noch die Endpunkte des Teilkreises mit dem Mittelpunkt zu verbinden. Schrift in die Grafik können wir mit dem ganz normalen PRINT-Befehl setzen, wobei allerdings die Position erst mit dem LOCATE-Befehl der Grafik angepaßt werden muß (LOCATE und PRINT werden im BASIC-Handbuch sehr gut beschrieben). Das Berechnen der Positionen ist eine mathematisch recht einfache Angelegenheit, die ich Ihnen überlasse. Experimentieren Sie mit Ihrem Wissen! Wer experimentiert, lernt schneller.

Wer nun wirklich experimentiert hat, wird gemerkt haben, daß eine sinnvoll eingesetzte Grafik nicht nur Sache des Beherrschens von Grafikbefehlen ist. Wie zum Beispiel findet man die richtigen Linienkoordinaten zur Verbindung des Mittelpunktes mit dem Ende des Teilkreises? Wie berechnet man die richtige Position für Beschriftung? Eine gehörige Portion Logik und einige Mathematik-Kenntnisse benötigt man schon auch. Und die Moral von der Geschicht: Üben, üben, üben! Programmierpraxis ist der beste Lehrmeister.

12.1.6 Sonstige Objekte

Was gibt es denn nun noch? Wenn Sie der Meinung sind, daß wir doch schon alles hinter uns haben, was es an geometrischen Objekten gibt,

liegen Sie falsch! Denn da gibt es noch Formen wie Spiralen, schief liegende Spiralen, Ovale, gebogene Linien und so weiter.

Alle diese Formen kann man mit einigen Tricks aus einzelnen Linien und Kreisen zusammensetzen. Die Spirale wird zum Beispiel aus lauter Teilkreisen zusammengesetzt. Ein Teilkreis muß immer jeweils da beginnen, wo der letzte aufhört, muß dann aber mit größerem Radius gezeichnet werden. Außerdem sollte der Streckungsfaktor (Aspect) entsprechend angepaßt werden. Bei einer Spirale von innen nach außen muß er langsam – in sehr kleinen Schritten – größer werden. Am besten hilft auch hier wieder Experimentierfreudigkeit. Natürlich kann man das Ganze aber auch mit einzelnen Punkten bewerkstelligen. Hierzu empfiehlt sich eine leicht abgewandelte Form des im ersten Buchteils besprochenen Kreis-Algorithmus. Nachteil dieser Lösung ist, daß die Geschwindigkeit dabei sehr stark abfällt.

Gebogene Linien können wir ebenfalls mit einer Abwandlung des Circle-Befehls darstellen. Noch besser eignet sich dazu allerdings ein sogenannter »Spline-Algorithmus«. Ein Spline-Algorithmus ist dazu gedacht, eine schön geschwungene Linie durch eine Folge von Punkten zu legen. Damit kann man zum Beispiel auch die Ecken von Polygonen abrunden, Kurven zeichnen und so weiter Jetzt zu erklären, wie so ein Spline-Algorithmus funktioniert, würde aber zu sehr auf mathematische Ergüsse hinauslaufen. Wer sich dafür interessiert, kann einen Spline-Algorithmus in Kapitel 21 der »Grundzüge der interaktiven Computergrafik« (von William Newman und Robert Sproul, erschienen bei McGraw-Hill) nachlesen. Das Buch befaßt sich außerdem mit komplexen 3D-Grafiken, Schattierungen, und so weiter. Jedem, der sich ernsthaft für diese Themen interessiert und mathematische Kenntnisse zumindest der mittleren Reife besitzt, ist es unbedingt zu empfehlen.

Schrägliegende Ellipsen werden übrigens von fast keinem einzigen Programm unterstützt, auch hierzu bräuchte man einen eigenen Algorithmus. Das einzige mir bisher bekannte Programm, das auch schiefliegende Ellipsen berücksichtigt, ist »Aegis Images« (siehe hierzu auch das Kapitel »Professionelle Grafiksoftware«). Weitere Objekte können aus Kombinationen von Linien, Teilkreisen und Spline-Kurven bestehen. Hier liegt es am Programmierer, den besten Effekt mit möglichst wenig Aufwand zu erzeugen.

Bisher sprachen wir die ganze Zeit von »Objekten«. Mit dem, was wir bisher gelernt haben, können wir bereits ein sogenanntes »objektorientiertes« 2D-CAD-Programm schreiben. Beispiel dafür ist ein in AmigaBASIC geschriebenes Demo zum Programm »MicroCAD«, einem CAD-Programm, das in der endgültigen Version in Forth geschrieben ist und sehr professionelle Leistungsmerkmale zeigt. Das Demo wurde von den Programmierern als Vorführ-Version schnell in BASIC entwickelt, bevor das endgültige Produkt geschrieben wurde. Es zeigt, daß man auch in BASIC durchaus professionelle Software entwickeln kann.

Objekte hin, Objekte her; bisher waren unsere Exkursionen nicht besonders farbenfroh. Das soll sich nun ändern – denn erst die Farbfähigkeiten machen den Amiga zu einem außergewöhnlichen Grafiksystem. Viele Computer hatten schon die Grafikauflösung des Amiga – aber noch kein Computer dieser Preisklasse konnte so farbenfrohe und damit detaillierte Grafik darstellen. Deshalb nun auf in die Welt der Farbe – im folgenden Abschnitt!

12.2 Von Farben, Screens und Windows

12.2.1 Screens und Farben

Wie Sie sicher wissen, verfügt der Amiga durch seine Systemsoft- und -Hardware über die Möglichkeit, sogenannte »virtuelle Bildschirme«, auch »Screens« genannt, zu eröffnen. Die Bildschirme werden als sogenannte »Playfields« dargestellt, also Grafikbereiche, die wiederum aus einzelnen Bitplanes zusammengesetzt sind. Wenn Sie den Hardwareteil aufmerksam gelesen haben, wissen Sie bereits, wie das genau funktioniert. Wenn Sie sich jetzt noch einmal den Workbench-Screen ansehen, in dem auch AmigaBASIC seine Windows eröffnet, werden Sie schnell merken, worauf ich hinauswill. Da der Workbench-Screen aus nur drei Bitplanes besteht, sind nicht mehr als vier Farben (inklusive Hintergrundfarbe) möglich. Damit wäre auch das BASIC auf diese doch sehr geringe Anzahl von Farben und auf die Grafikauflösung der Workbench (640 mal 200) begrenzt – gäbe es nicht den SCREEN-Befehl.

SCREEN Nummer,Breite,Höhe,Tiefe,Modus

Auf die einzelnen Angaben (Parameter) kommen wir gleich zu sprechen. Das erste, was der SCREEN-Befehl bei seiner Ausführung tut, ist das Bereitstellen des notwendigen Speicherplatzes für die Grafik. Anschließend wird der Screen mit den gewünschten Parametern dargestellt.

Das Parameter »Nummer« ist die Zahl des Bildschirms. AmigaBASIC kann bis zu vier Screens öffnen. Die Screen-Nummer wurde eingebaut, damit wir für die Ausgabe von Grafiken eine Referenznummer haben, auf welchen Screen die Grafik ausgegeben werden soll. AmigaBASIC kann Grafiken grundsätzlich nur in Windows darstellen, wir müssen also in jedem Screen auch noch ein Window öffnen. Die Screen-Nummer dient aber auch dazu, Windows entsprechenden Screens zuzuordnen, also beispielsweise Window 1 in Screen 1, Window 2 und 3 in Screen 2 und so weiter. Vergleichen Sie dazu bitte auch das Kapitel »Windows«.

Die Parameter »Breite« und »Höhe« geben die Größe des benutzbaren Teils des Screens an. Soll beispielsweise nur der linke obere Teil des Bildschirms verwendet werden und der Rest die Hintergrundfarbe annehmen, setzen wir für diese Parameter Werte ein, die kleiner als die Auflösung sind. Der Wert für Breite kann zwischen 1 und 640 liegen, die Höhe kann zwischen 1 und 400 liegen. Allerdings sollte man damit nicht zuviel herumexperimentieren, denn bei diesem Befehl ist AmigaBASIC nicht ganz fehlerfrei. Bevor Sie ein Programm starten, dessen SCREEN-Parameter für Breite und Höhe nicht der Auflösung entsprechen, speichern Sie es lieber erst einmal ab. Manchmal hat man wider Willen ganz andere Dinge auf dem Bildschirm als eigentlich beabsichtigt, manchmal meldet sich der berüchtigte Guru (System-Alert) zu Wort. Sollte es funktionieren, bietet diese Methode einen entscheidenden Vorteil: Sie sparen Speicherplatz. Wenn wir nur die obere Hälfte eines Bildschirms benutzen (weil darunter ein anderer beginnt), ist es Unsinn, Speicherplatz für die ohnehin bedeckte untere Hälfte freizuhalten.

Der Parameter »Tiefe« gibt die Tiefe in Bitplanes an. Die Tiefe bestimmt damit auch die Anzahl der Farben. Folgende Möglichkeiten stehen zur Auswahl:

Anzahl Bitplanes (Tiefe)	Anzahl der Farben
1	2
2	4
3	8
4	16
5	32

Tabelle 12.1: Zusammenhang zwischen Screentiefe und Farben

Und nun zum letzten Parameter, dem Grafik-Modus. Diese Zahl gibt an, welche Bildauflösung der neue Screen besitzen soll. Je höher die Zahl für den Grafikmodus ist, desto mehr Speicher wird pro Bildschirm verbraucht.

Modus	Bezeichnung	Auflösung
1	LoRes, NonInterlaced	320 x 200
2	HiRes, NonInterlaced	640 x 200
3	LoRes, Interlaced	320 x 400
4	HiRes, Interlaced	640 x 400

Tabelle 12.2: Auswahl des Grafik-Modus

LoRes bedeutet beim Amiga immer 320 Punkte horizontale Auflösung, HiRes bedeutet 640 Punkte horizontale Auflösung. Interlace bedeutet eine Verdoppelung der normalen vertikalen Auflösung von 200 auf 400 durch Überlagern zweier Halbbilder. Wie Interlace genau funktioniert, wurde bereits im Hardwareteil ausführlich geschildert.

Leider verfügt der Screen-Befehl nicht über die Möglichkeit, die Position des Screens am Bildschirm und seine relative Priorität zu anderen Bildschirmen anzugeben. Sobald der Befehl ausgeführt ist, erscheint der Bildschirm vor allen anderen Screens, mit der y-Position 0. Das Auf- und Abschieben des Bildschirms muß man also dem Benutzer überlassen. Schade, denn damit hätte sich eine interessante Möglichkeit für eine Erstellung von Bildschirmmasken ergeben; durch die Verteilung auf verschiedene Screens lassen sich an verschiedenen y-Positionen des Bildschirms unterschiedliche Auflösungen und unterschiedliche Farben darstellen. Durch die Höhe-

und Breite-Parameter ließe sich der nicht benötigte Speicherplatz für die verdeckten Teile der Einzelscreens einsparen.

Was ich meine, werden Sie am folgenden kleinen Beispielprogramm sehen, das den Screen-Befehl und seine Wirkungsweise demonstriert. Zum Verständnis des Listings benötigen Sie allerdings noch etwas Wissen über Farben und Windows. Die Windows werden Sie im nächsten Kapitel näher kennenlernen. Geben Sie sich also vorerst bitte damit zufrieden, daß Window 1 im ersten Screen und Window 2 im zweiten Screen geöffnet werden. Welches Aussehen die Farben haben, setzen wir durch den PALETTE-Befehl fest.

PALETTE Farbnummer,Rotanteil,Grünanteil,Blauanteil

Die Farbnummer ist eine Zahl von 0 bis 31, bei Screens mit weniger Bitplanes ist die höchstmögliche Farbnummer natürlich entsprechend niedriger (gemäß der oben aufgeführten Tabelle). Die Rot- Grün- und Blauanteile jeder Farbe werden mit Zahlen von 0 bis 1 angegeben. Beispiel: PATTERN 1,1,1,0.13 ergibt ein leuchtendes Gelb für Farbe 1. Der PALETTE-Befehl gilt jeweils nur für den gerade aktiven Screen. Das ist auch in dem folgenden Listing sehr deutlich zu sehen. (Bitte beachten Sie beim Eintippen die richtige Anzahl von Kommata und Punkten.)

```
SCREEN 1,320,200,2,1
 WINDOW 1,,(20,20)-(290,100),,1
  PALETTE 0,.2,.2,.2
  PALETTE 1,.4,.4,.4
  PALETTE 2,.8,.8,.8
 WINDOW OUTPUT 1
   FOR i = 1 TO 290 STEP 3
    LINE (i,0)-(i,150),1
    LINE (i+1,0)-(i+1,150),2
    LINE (i+2,0)-(i+2,150),3
   NEXT
SCREEN 2,320,200,2,1
 WINDOW 2,,(20,20)-(290,100),,1
  PALETTE 0,1,.3,1
  PALETTE 1,.3,.2,.2
  PALETTE 2,1,.4,0
```

```
WINDOW OUTPUT 2
  FOR i =  1TO 290 STEP 3
   LINE (i,0)-(i,150),1
   LINE (i+1,0)-(i+1,150),2
   LINE (i+2,0)-(i+2,150),3
  NEXT
```

Wenn Sie das Programm starten, werden Sie sehen, wie zuerst ein Screen mit den zuerst gewählten Farben geöffnet wird. In diesem Screen ist ein Window, in das verschiedenfarbige Linien gezeichnet werden. Bei Aufruf des zweiten Screen-Befehls verschwindet der erste sofort wieder, und es erscheint dort der zweite Bildschirm mit einem gleichartigen Window und gleichartiger Grafik, allerdings in anderen Farben. Ziehen Sie nun mit der Maus den vordersten Screen (Screen 2) bis fast nach unten. Dahinter erscheint Screen 1, für den nach wie vor die alten Farben gelten. Ziehen Sie diesen Screen auch etwas nach unten, und schon erscheint dahinter der Workbench-Screen.

Klicken Sie mit der Maus in das Window des mittleren Screens (der graue Bildschirm). Daraufhin erscheint das vertraute OK des Output-Windows von AmigaBASIC. Tippen Sie nun den folgenden Befehl ein:

```
SCREEN CLOSE 2
```

Und – schwupp! – verschwindet der unterste Bildschirm. Mit *SCREEN CLOSE Nummer* wird der entsprechende Screen abgeschaltet und der Speicher freigegeben. Wenn Sie einen Screen nicht benötigen, schalten Sie ihn bitte immer mit SCREEN CLOSE ab! Öffnen Sie nämlich zuviele Bildschirme (zwei HiRes-Interlace-Schirme und der Workbench-Screen, dazu vielleicht ein paar Icons auf der Workbench, das reicht schon), erscheint der bereits oft zitierte Guru und Sie müssen den Amiga neu starten – AmigaBASIC ist leider nicht ganz absturzfrei.

Warum wirkt nun der Palette-Befehl nur auf einzelne Screens? Wir haben doch nur 32 Farbregister in der Hardware des Amiga, wie können wir also in einem Screen 32 Farben, in einem anderen Screen ganz andere 32 Farben verwenden? Die Antwort fällt nicht ganz einfach aus. Dieser Farbwechsel hängt mit den Grafikroutinen des Betriebssystems zusammen. Die Farben werden von AmigaBASIC in Farbtabellen eingebunden, die von den Grafikroutinen (Grafiklibrary

und Intuition) verwendet werden. Wie Sie vielleicht schon im Grundlagenteil gelesen haben, wird das Bild von links oben nach rechts unten aufgebaut, indem ein Rasterstrahl zeilenweise über das Bild läuft und die entsprechende Farbe eines Pixels anzeigt. Die Systemsoftware des Amiga läuft quasi mit diesem Rasterstrahl mit. Ist der Strahl an einer Stelle angekommen, an der ein neuer Screen beginnt, holt sich das System die Farben für den nächsten Screen aus diesen Farbtabellen und schreibt sie in die Farbregister. Aus diesem Register holt sich die Display-Hardware die Farbe, die über den Rasterstrahl ausgegeben wird.

Wer vom guten alten C64 oder 8-Bit-Atari kommt, war gewohnt, gleich direkt mit POKE-Befehlen die Farben in den Farbregistern zu ändern. Das geht am Amiga nicht. Versuchen Sie es ruhig selbst. Mehr als ein kurzes Flackern bekommen Sie nicht zu sehen, da sich das Betriebssystem ja nach spätestens einer 30stel-Sekunde die Farbe aus den Farbtabellen holt und sie wieder in die Farbregister schreibt.

Ein weiterer Befehl zur Farbkontrolle ist der COLOR-Befehl.

COLOR Vordergrund,Hintergrund

Hierbei ist »Vordergrund« die Farbe, mit der Schrift dargestellt wird, aber auch Grafik, wenn der Farbparameter des Grafikbefehls fehlt, und »Hintergrund« ist die Nummer der Farbe, die den Grafikhintergrund füllt. Beide Parameter können Werte zwischen 0 und 31 annehmen und »zeigen« auf die entsprechende, mit dem PALETTE-Befehl bestimmte, Farbnummer.

12.2.2 Windows

Mit den Windows oder Bildschirmfenstern ist dem Computer bekanntlich die Benutzerfreundlichkeit eingehaucht worden. Doch die muß zuerst einmal programmiert werden.

Windows sind ein wichtiges Grafikelement. Alle Grafiken in AmigaBASIC können zudem nur in Windows ausgegeben werden. Für jede Grafik müssen wir daher ein Window öffnen. Die Koordinaten (0,0) für eine Grafik liegen immer in der linken oberen Ecke des Windows. Koordinaten haben nichts mit dem Screen zu tun. Das können Sie am letzten Beispiel ausprobieren, indem Sie im Window-Befehl ganz einfach den y-Wert ändern (zweiter Wert 20 innerhalb der

Zeile). Die Grafik beginnt, ganz gleich wo das Window ist, immer links oben im Window. Zum Öffnen eines Windows ist der WINDOW-Befehl vorgesehen.

WINDOW Nummer ',Name' ',Rechteck' ',Typ' ',Screennummer'

Auch hier sind die nicht unbedingt benötigten Parameter wieder in Hochkommata gesetzt. Es genügt also schon *WINDOW 1*, um ein Fenster zu öffnen. Mit diesem Befehl machen wir das BASIC-Output-Window zu unserem Grafik-Window. WINDOW 1,"Tolles Window" macht den Text »Tolles Window« zum Titel des neuen Fensters.

Der Parameter »Rechteck« muß wie üblich in der Form (x1,y1)-(x2,y2) angegeben werden und bestimmt die Größe des Fensters. *WINDOW 1,,(10,10)-(50,50)* öffnet zum Beispiel ein Window, dessen linke obere Ecke am Punkt (0,0) des Screens liegt und das 40 Punkte breit und 40 Punkte hoch ist. Bitte beachten Sie auch hier wieder die Regel, Kommata auch dann zu setzen, wenn der Parameter dazwischen fehlt. In diesem Beispiel fehlte der Name, also benötigen Sie nach der Windownummer zwei Kommata.

Der Windowtyp setzt fest, was das System und der Benutzer mit dem Window machen können. Die hier angegebene Zahl entspricht einer Folge von Bits, von denen jedes für eine bestimmte Eigenschaft des Fensters steht. Sie können diese Zahl errechnen, indem Sie einfach die Bitmasken der gewünschten Eigenschaften addieren.

Bitmaske (dezimal)	Bedeutung
1	Größenveränderungs-Gadget
2	Bewegungs-Gadget
4	Vordergrund/Hintergrund-Gadget
8	Schließ-Gadget
16	Smart Refresh

Tabelle 12.3: Bedeutung der Bits in Typ eines neuen Fensters

Der Wert für ein Fenster, dessen Größe vom Benutzer verändert werden kann und das auch bewegt werden darf, ist also 3 (1+2). Je nach dem hier festgelegten Wert für den Windowtyp erscheinen im Window die entsprechenden Gadgets.

Sicher werden Sie sich schon gefragt haben, was es mit »Smart-Refresh« auf sich hat. Smart Refresh ist eine Fähigkeit des Amiga-Systems, selbsttätig die Wiederherstellung eines vorübergehend verdeckten Windowinhaltes zu übernehmen. Das Gegenteil dazu ist der Simple-Refresh. Wird ein Simple-Refresh-Fenster einmal von einem anderen verdeckt, ist sein Inhalt verloren und muß vom Programm wiederhergestellt werden. Die Smart-Refresh-Technik speichert den verdeckten Teil eines Fensters in einem Puffer. Auch alle Grafikbefehle, die den verdeckten Teil des Fensters betreffen, gehen in diesen Puffer. Legt man ein verdecktes Fenster wieder frei, wird dieser Puffer in den Bildschirm kopiert und es sieht so aus, als wäre nie etwas gelöscht worden.

All diese Zwischenspeicherungsaufgaben werden automatisch vom Betriebssystem des Amiga durchgeführt. Im AmigaBASIC unterscheiden wir nur zwischen der Simple-Refresh-Technik und der Smart-Refresh-Technik. Das Betriebssystem des Amiga kennt noch eine dritte Technik. Ausführlich erklärt habe ich das bereits im Kapitel 14.13.1 des Amiga-Handbuchs von Markus Breuer, an dessen Hardware- und Grafikteil ich beteiligt war. Sollten Sie sich dafür interessieren, schlagen Sie bitte in diesem Buch nach.

Und nun schließlich zum letzten Parameter: Die Screennummer im Windowbefehl gibt an, auf welchem Screen das Window erscheinen soll. Geben wir beispielsweise WINDOW 1,,,,1 (vier Kommata!) ein, wird das BASIC-Output-Window auf Screen 1 gelegt.

Sind mehrere Windows geöffnet, wollen wir natürlich unsere Grafiken nicht nur auf eines dieser Windows beschränken. Mit dem Befehl WINDOW OUTPUT können wir bestimmen, in welches Fenster Grafikbefehle und PRINT-Anweisungen gehen.

WINDOW OUTPUT Nummer

Dieser Befehl lenkt alle Grafik- und Textausgaben, die danach folgen, auf das Window mit der angegebenen Nummer um. Sollen in zwei Windows gleichzeitig Grafiken entstehen, müssen wir also mit dem WINDOW-OUTPUT-Befehl immer zwischen den verschiedenen Windows hin- und herschalten. Sehr gut gelungen ist das im Programm »Amigademo«, das auf der Extras-Diskette in der Schublade »BASICDemos« abgelegt ist. Sehen Sie sich einmal das Listing an; es ist einfach aufgebaut und gut dokumentiert.

Offene Fenster können natürlich auch geschlossen werden. Dazu dient die WINDOW-CLOSE-Anweisung.

WINDOW CLOSE Windownummer

Der Speicher, der für das Window belegt wurde, wird dadurch freigegeben. Das Window verschwindet mitsamt Inhalt bei Aufruf des CLOSE-Befehls.

Was tun wir nun, wenn der Benutzer mit den Gadgets das Window verändert? Wenn wir zum Beispiel einen Ball immer bis zum Rand eines Windows hüpfen lassen wollen, ganz gleich wie groß es gerade ist, müssen wir die Größe des Windows abfragen. Und damit sind wir wieder beim Schritt von der statischen zur interaktiven Computergrafik, sprich bei der Kommunikation mit dem Benutzer. Zur Feststellung wichtiger Daten, denen wir unsere Grafiken anpassen wollen, benötigen wir glücklicherweise nur eine Funktion.

WINDOW (n)

Je nach Argument »n « erhalten wir dadurch eine Reihe von ganz unterschiedlichen Informationen über das aktuelle Fenster beziehungsweise das Output-Fenster.

Argument	Bedeutung
0	Windownummer des aktiven Fensters
1	Windownummer des Output-Fensters
2	Breite des Output-Fensters
3	Höhe des Output-Fensters
4	X-Position des Textcursors im Output-Fenster
5	Y-Position des Textcursors im Output-Fenster
6	Anzahl Farben im Output-Fenster
7	Zeiger auf Window-Record des Output-Fensters
8	Zeiger auf RastPort-Record des Output-Fensters

Tabelle 12.4: Mögliche Ergebnisse der WINDOW-Funktion

Die letzten beiden Möglichkeiten brauchen Sie vorerst nicht zu beachten. Sie sind für sehr fortgeschrittene Programmierer. Wir werden uns diesem Problem später noch widmen.

Der Unterschied des mit n = 0 zu dem mit n = 1 abgefragten Windows ist der folgende: Das aktive Fenster ist das zuletzt mit der Maus angeklickte Output-Fenster. Wollen wir den Benutzer mit mehreren Windows arbeiten lassen, müssen wir ja auch feststellen, in welches er geklickt hat, um daraufhin unsere Ausgabe in dieses Fenster umzulenken. Das Output-Fenster ist hingegen das Fenster, in das die Grafik- und PRINT-Befehle im Augenblick gesendet werden.

Die Abfrageparameter 3 und 4 sind eigentlich schon aus der Tabelle ersichtlich. Damit stellen wir eine eventuell mit der Maus veränderte Größe des Windows fest. Das tut auch das AmigaDemo auf der Extras-Diskette mit seinen Windows. Abfrageparameter 5 und 6 geben die x- und y-Koordinate an, wo der nächste Buchstabe per PRINT-Befehl dargestellt werden würde.

Damit hätten wir die Windows auch schon abgehandelt. Was Sie daraus machen, ist Ihre Sache. Mit dem bisherigen Wissen über Screens, Farben, Grafikobjekte und andere Elemente der Grafik ist es Ihnen nun schon möglich, professionelle Grafiksoftware zu schreiben. Das wird Ihnen auch unser kleines 3D-Programm im Beispiele-Kapitel zeigen. Mehr als die bisher genannten Befehle werden dort nicht verwendet. Zum Programmieren eines Profi-Malprogrammes fehlt Ihnen nur noch die Abfrage von Maus und Menüs mittels der MOUSE und MENU-Kommandos. Dazu schlagen Sie bitte in den einschlägigen BASIC-Handbüchern nach. Der Rest ist eine Frage von Programmiermethoden und Algorithmen, nicht jedoch der Grafikbefehle. Ein Programm mit den Fähigkeiten von DeluxePaint läßt sich auch durchaus in BASIC schreiben, ist dann allerdings nicht so schnell wie das Original.

12.3 Flächenbefehle

Flächen sind Teile einer Grafik. Unter Flächenbefehlen verstehen wir alles, was Flächen erstellt, ausfüllt oder bewegt. So könnten wir beispielsweise auch die Polygonbefehle (AREA, AREAFILL) als Flächenbefehle auffassen. Sie sind es auch, aber sie sind zugleich

geometrische Objekte. Daß ich die Polygonbefehle nicht unter die Flächenbefehle gestellt habe, war rein willkürlich.

Flächen entstehen durch das Ausfüllen eines beliebigen geometrischen Objektes. Wir geben einen Punkt inmitten eines Kreises, Rechtecks, einer Ellipse oder einer anderen Form an. Alle Punkte innerhalb dieser Form können dann mit einem einzigen Befehl ausgefüllt werden.

PAINT 'STEP' (x,y) ',Füllfarbe' ',Grenzfarbe'

Haben wir zum Beispiel ein Rechteck mit *LINE (10,10)-(20,20),b* gezeichnet, füllt der Paint-Befehl *PAINT (11,11)* dieses Rechteck aus. Mit der STEP-Option kann ab einer Position gefüllt werden, die x Punkte rechts und y–Punkte unterhalb des zuletzt gezeichneten Punkt liegt.

Die Füllfarbe bestimmt die Farbe, mit der die Fläche gefüllt wird, die Grenzfarbe gibt die Farbe der Punkte an, die den »Farbfluß« begrenzen sollen. So können auch bereits mehrfarbig gefüllte Flächen neu übermalt werden, ohne mehrmals den PAINT-Befehl aufzurufen. Wird weder die Füll- noch die Grenzfarbe angegeben, wird für beide Werte die durch den COLOR-Befehl bestimmte Vordergrundfarbe verwendet.

Es gibt mehrere Möglichkeiten, Flächen im Speicher abzulegen und die in diesen Flächen liegenden Grafiken nachher wiederzuverwenden, ohne sie extra noch einmal zeichnen zu müssen. Die eine Möglichkeit wäre es, jeden Punkt innerhalb einer Fläche mit POINT abzufragen und den Farbwert für jedes Pixel einer Variable zuzuweisen. Das ist allerdings so langsam, daß man besser dieselbe Grafik nochmal zeichnen lassen kann. Die schnellere Variante ermöglicht der GET-Befehl.

GET (x1,y1)-(x2,y2),Feldvariable '(index ',index,index,...')'

Dieser Befehl erlaubt allerdings nur die Entnahme rechteckiger Grafiken aus einem Bild. Mit dem GET-Befehl wird das Rechteck in dem Bereich (x1,y1)-(x2,y2) in das Array »Feldvariable« gespeichert. Die Feldvariable muß vorher in der passenden Größe (mit dem DIM-Befehl) dimensioniert worden sein. Die verschiedenen Index-Parameter erlauben die Speicherung mehrerer Objekte in einem multidimensionalen Grafikarray. Das erlaubt bei der Ausgabe der

zwischengespeicherten Grafik durch den PUT-Befehl eine schnelle Ausgabe unterschiedlicher Bewegungsphasen eines Objektes innerhalb einer Schleife. (Siehe dazu auch das Kapitel »Animationstechniken«.)

Der benötigte Speicherplatz (in Bytes) für das Grafikarray »Feldvariable« errechnet sich wie folgt:

Größe = 6+((y2-y1+1)*2*INT((x2-x1+16)/16)*T

Dabei ist »T« die Tiefe der Grafik (Anzahl der Bitplanes). Da ein INTEGER-Wert 2 Bytes Speicherplatz benötigt, teilen wir den oben herausgekommenen Wert durch 2. Damit bekommen wir heraus, auf wieviel Elemente wir ein Integer-Array dimensionieren müssen, in das wir die Grafik ablegen. Soll die Grafik in Single-Precision-Variable gespeichert werden, teilen wir das Ergebnis der obigen Formel durch vier, da ein Single-Precision-Wert 4 Bytes Speicherplatz benötigt. In Stringvariablen schließlich dürfen keine Grafiken gespeichert werden!

Eine auf diese Weise gespeicherte Grafik können wird dann mit dem PUT-Befehl wieder auf den Bildschirm bringen.

PUT (x,y),Feldvariable '(index,index....)' ',Aktionsverb'

Dabei ist (x,y) die linke obere Ecke der auf den Bildschirm plazierten Grafik. Die Index-Parameter sind bereits im Zusammenhang mit dem GET-Befehl besprochen worden. Das »Aktionsverb« kann PSET, PRESET, AND, OR und XOR heißen. Die folgende Tabelle gibt Aufschluß über die einzelnen Bedeutungen der verschiedenen Aktionsverben.

Aktionsverb	Bedeutung
PSET	Ersetzt den alten Bildschirminhalt durch die gespeicherte Grafik
PRESET	Ersetzt den alten Bildschirminhalt durch ein Negativ der gespeicherten Grafik
AND	Verknüft den alten Bildschirminhalt und die gespeicherte Grafik mit einem logischen UND
OR	Verknüft den alten Bildschirminhalt und die gespeicherte Grafik mit einem logischen ODER
XOR	Verknüft den alten Bildschirminhalt und die gespeicherte Grafik mit einem exklusiven ODER

Tabelle 12.5: Mögliche Wirkungen des PUT-Befehls

Wird also beispielsweise der Befehl *PUT (10,10),a,OR* gegeben, wird die Grafik über den Hintergrund kopiert, ohne den Inhalt des Hintergrunds an den Stellen zu verändern, wo die gespeicherte Grafik die Hintergrundfarbe enthält. Bei AND wird ein Grafikpunkt der PUT-Grafik nur dann auf den Hintergrund gesetzt, wenn dort bereits ein Punkt ist. Wird kein Aktionsverb angegeben, wird automatisch XOR als Aktionsverb angenommen.

Das Demo »Picture« in der BASICDemos-Schublade zeigt deutlich die Funktionsweise der GET- und PUT-Befehle. Sehen Sie sich das Listing einmal an, es ist relativ kurz.

Flächen bewegen kann man auch ohne den Speicherplatzaufwand, den uns die GET/PUT-Kombination auferlegt. Der SCROLL-Befehl erlaubt das Bewegen einer Grafik innerhalb eines rechteckigen Bereiches x1 bis y2:

SCROLL (x1,y1)-(x2,y2),delta-x,delta-y

Dabei wird nur die Grafik innerhalb des Bereiches (x1,y1)-(x2,y2) um »delta-x« Punkte nach rechts und um »delta-y« nach unten verschoben. (Soll nach links oder oben gescrollt werden, müssen negative Werte für delta-x und delta-y eingesetzt werden.) Außerhalb dieses Bereichs treten keine Veränderungen auf. Beim Scrollen einer Grafik aus diesem Bereich heraus verschwindet sie einfach. Das bietet natürlich eine effektvolle Methode, Schriftzüge oder Titelbilder verschwinden zu lassen (siehe auch das Kapitel "Animation"). Es

handelt sich hierbei um sogenanntes HiRes-Scrolling, das man nicht mit der Fähigkeit der Amiga-Hardware zum Playfield-Scrolling verwechseln darf (die leider von BASIC nicht unterstützt wird).

Mit dem Befehl SCROLL sind wir nun schon wieder beim nächsten Punkt gelandet: Die Animation. Wir sprechen von animierter Grafik dann, wenn Grafiken bewegt werden oder sich in schneller Folge ändern. Dazu gibt es mehrere Möglichkeiten, die im nachfolgenden Kapitel ausführlich beschrieben werden.



13

Animation in AmigaBASIC

13.1 Sprites und BOBs – die Animationskünstler

Die Animationsfähigkeiten des Amiga sind das, was diesen Computer für viele so attraktiv macht. Der Schlüssel zu Computeranimation sind in erster Linie bewegte Objekte. Der Amiga kennt sogar mehrere verschiedene Typen solcher Objekte.

13.1.1 Die Objekte

Der Amiga kennt im wesentlichen drei verschiedene Arten bewegter Objekte (von denen es allerdings noch eine Reihe verschiedener Abwandlungen gibt):

• Sprites, durch Hardware über das normale Bild gelegte Objekte.

• BOBs (**B**litter **Ob**jects), per Software erzeugte spriteähnliche Objekte. Die Betriebssystemsoftware des Amiga nutzt zur schnellen Animation dieser Objekte die Fähigkeiten des Blitter-Chips, logische Verknüpfungen und schnellen Datentransfer im Speicher durchzuführen.

• AnimObjects, Kombinationen von Sprites oder AnimObjekts mit spezieller Systemsoftware. Die Animationsbibliotheken des Amiga erlauben es, Sprites und Bobs mit den gleichen Routinen zu behandeln. Das wiederum macht sich das BASIC zunutze und gibt uns die Möglichkeit, Sprites und BOBs mit denselben Befehlen anzusprechen.

13.1.2 Objektfestlegung und -bewegung

In AmigaBASIC legen wir die Eigenschaften eines Objekts – gleich ob Sprite oder BOB – in Form einer Stringvariable fest. Der String sagt dem BASIC-Interpreter, wie groß das Objekt ist, ob es ein Sprite oder BOB ist, wieviel und welche Farben es beinhaltet und natürlich wie es aussieht. Wie dieser String genau aufgebaut ist, brauchen wir jedoch nicht zu wissen. Der Amiga BASIC beiliegende Objekteditor nimmt uns nämlich diese Lernarbeit ab. Alles, was wir tun müssen, ist es, das Objekt zu entwerfen. Das fertige Objekt wird vom Editor automatisch in der Form gespeichert, die wir als String später von Diskette holen und als Sprite oder BOB verwenden können. In der Regel greifen deswegen BASIC-Programme, die Objektanimation beinhalten, nach dem Programmstart auf die Diskette zu. Lassen Sie also nach dem RUN-Befehl immer die entsprechende Diskette im Laufwerk. (Übrigens: Wer wissen will, wie die AnimObject beschreibende Stringvariable aussehen muß, sollte sich das Listing des Objekteditors einmal ansehen. Es ist sehr gut dokumentiert.)

Starten Sie nun den Objekteditor entweder aus der Workbench (OBJEDIT in der BASIC-Demos-Schublade) oder vom BASIC-Interpreter direkt mit *LOAD "BASICdemos/Objedit"* und RUN.

Das Programm fragt Sie zuerst, ob Sie Sprites oder BOBs editieren wollen. Jeder der Objekttypen hat seine Vor- und Nachteile, die teilweise von der Hardware, teilweise aber auch vom BASIC-Interpreter abhängig sind. BOBs sind langsamer als Sprites, dafür aber ist ihre Größe nur abhängig vom freien Speicher, wogegen die Sprites nur 16 Pixels breit sein dürfen. Bobs können in beliebiger Anzahl angezeigt werden und genauso viele Farben enthalten wie der Bildhintergrund. Sprites sind auf drei (beziehungsweise vier) Farben limitiert. Es können nur vier Sprites mit unterschiedlichen Farben zur selben Zeit auf derselben Höhe (auf denselben Rasterzeilen) erscheinen. Für farbenfrohe Animation empfehlen sich also BOBs. Geht es mehr um Geschwindigkeit, sollte man Sprites benutzen. Für detailreiche Animation, die auch ein gewisses Maß an Geschwindigkeit mit sich bringen muß, empfiehlt sich eine (sinnvolle !) Kombination aus Sprites und BOBs.

Tippen Sie für unser Beispiel nun eine Null für BOBs ein. Editieren Sie dann Ihr eigenes Objekt, wie Sie es gern hätten. Für unser Beispiel genügt ein kleines Kunstwerk (bezogen auf die Größe, nicht

auf künstlerisches Talent). Wie Sie sehen, ist die Farbwahl auf vier Farben beschränkt. Wenn Sie einen 512K-Amiga haben, können Sie natürlich mehr Farben benutzen. Dazu müssen Sie im Objekteditor drei Zeilen ändern. (Diese Zeilen sind im Listing dokumentiert.) Alles, was Sie dabei machen müssen, ist das Entfernen von drei Hochkommas (die diese Zeilen als Kommentare kennzeichnen.)

Speichern Sie das Objekt nun unter einem beliebigen Namen ab. Diesen Namen benötigen wir später wieder. (Dann nämlich, wenn wir das Objekt im eigenen Programm verwenden wollen.) Der erste Schritt zur eigenen Animation ist eine Routine, die das Objekt von Diskette holt. Zum Aktivieren des Objektes benützen wir dann den BASIC-Befehl OBJECT.SHAPE.

OBJECT.SHAPE Nummer, Stringdefinition.

Die Stringdefinition des Objektes müssen wir von Diskette lesen. Natürlich könnten wir den String auch im Programmtext als Datazeilen ablegen, was aber den Nachteil hat, daß wir dafür dann dreimal Speicher verbrauchen: Einmal für den Programmtext, der den String beschreibt, einmal für den String, den wir aus den Datazeilen zusammenfügen, und einmal für das Objekt selbst. Ein Laden des Strings von Diskette empfiehlt sich also. Die folgenden drei Zeilen geben ein Beispiel, wie das Laden einer Objekt-Beschreibung aussehen könnte. Nach diesen Befehlen ist das Objekt sofort für Animation verfügbar.

```
OPEN "Name" FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
CLOSE 1
```

Wollen wir mehrere Objekte, die gleich aussehen, benutzen, wenden wir die zweite Form des OBJECT.SHAPE-Befehls an.

```
OBJECT.SHAPE 1,2
```

Dieser Befehl legt eine Kopie des Objects Nummer 1 an und weist ihm die Nummer 2 zu. Das zweite Objekt braucht dadurch keinen zusätzlichen Speicherplatz; beide Objekte greifen auf denselben Stringausdruck zu. Nun haben wir zwei Objekte, die wir über den Bildschirm sausen lassen können. Bevor wir aber überhaupt etwas

bewegen, müssen wir zuerst die Startposition der Bewegung festlegen. Das geschieht durch die Befehle OBJECT.X und OBJECT.Y.

OBJECT.X Nummer,Wert
OBJECT.Y Nummer,Wert

Die Positionsangabe kann einen Wert zwischen -32768 und 32767 annehmen; das ist allerdings Unsinn, denn der Bildschirm ist nicht so groß. Arbeiten wir mit dem normalen BASIC-Bildschirm (der dem Workbench-Screen entspricht), muß der x-Wert zwischen 0 und 639 liegen, der y-Wert muß zwischen 0 und 199 (beziehungsweise 255 bei Verwendung der Workbench 1.2) liegen. Wird durch den SCREEN-Befehl eine andere Bildschirmauflösung gewählt (, darf der x-Wert bei Low-resolution höchstens 319 sein, der y-Wert kann im Interlace-Modus bis zu 399 (beziehungsweise 511) betragen.

Allein durch das Festlegen von Koordinaten mit diesem Befehl könnten wir bereits eine Animation erzeugen. Das empfiehlt sich aber nicht, denn die Koordinatenänderung müßte ständig durch das BASIC-Programm berechnet werden. Wir wollen, während sich das Objekt bewegt, unser BASIC-Programm weiterlaufen lassen. Durch die Animationsbefehle des AmigaBASIC ist das möglich. Wir brauchen lediglich die Geschwindigkeit und eine Bewegungsrichtung festzusetzen, der Rest geht (fast) von ganz allein. Das leisten die Befehle OBJECT.VX und OBJECT.VY (siehe unten). Zunächst aber noch einmal zu den Objektkoordinaten. Die Befehle OBJECT.X und OBJECT.Y können nämlich noch auf eine zweite Art (als Funktionen) verwendet werden. Die Syntax dafür lautet wie folgt:

x = OBJECT.X (Objektnummer)
y = OBJECT.Y (Objektnummer)

Dadurch können wir jederzeit kontrollieren, wo sich unser Objekt im Augenblick befindet (zum Beispiel interessiert es den Autor eines Space-Invader-Spiels, wie tief das unterste Alien bereits steht). Haben wir unser Sprites erst einmal positioniert, sollten wir es sichtbar machen, quasi »anschalten».

OBJECT.ON 'Nummer ',Nummer....' '

Der OBJECT.ON-Befehl bewirkt, daß unser kleines Kunstwerk (Sprite oder BOB) sichtbar ist. Werden die Nummern weggelassen,

auch angeschaltet ist (OBJECT.ON). Die Bewegung können wir mit dem Gegenbefehl OBJECT.STOP zum Erliegen bringen.

OBJECT.STOP 'Nummer ',Nummer,.."

13.1.3 Krawumm – wir stoßen zusammen

Wenn wir beim Beispiel des Autorennens bleiben, kann dabei ja auch noch einiges passieren, was uns als Programmierer betrifft; Verkehrsunfälle sind ja bei gerade frisch programmierten Führerschein-Neulingen nicht selten. Bevor wir unsere »Krawumm«-Unterroutine aufrufen, sollte aber erstmal festgestellt werden, ob überhaupt eine Kollision stattfand – die Kfz-Versicherung wollen wir doch nicht übers Ohr hauen, oder?

Immer, wenn eine Kollision auftritt, stoppt AmigaBASIC die Bewegung der beiden kollidierenden Objekte. Dasselbe passiert, wenn ein Objekt die Fenstergrenzen berührt. Deswegen ist es oft notwendig, die Bewegung mit OBJECT.START wieder in Gang zu setzen. In vielen Fällen wird man vielleicht aber auch eine wunderschöne Explosion in Szene setzen. Beides kann man aber nur tun, wenn BASIC eine Kollision erkennt. Dafür gibt es das sogenannte »Event-Trapping« (vielleicht allzu wörtlich übersetzbar mit »Ereignisfangen«). Der Event, also das Ereignis, ist in unserem Falle eine Kollision. Das Event-Trapping wird durch den Befehl COLLISION ON aktiviert.

COLLISION ON

Es kann jederzeit mit COLLISION OFF wieder abgeschaltet werden – worauf Kollisionen zwar noch vorkommen können, aber nicht mehr erkannt werden.

COLLISION OFF

Ein weiteres Element des Kollisions-Event-Trapping ist der Befehl COLLISION STOP. Er sorgt dafür, daß die Kollisionen zwar registriert und in einer »Event queue« (die bis zu 16 Kollisionen aufnehmen kann) gespeichert werden. Er leitet diese Information aber zunächst nicht an das Programm weiter.

COLLISION STOP

Sind Kollisionen nicht durch COLLISION OFF oder COLLISION STOP abgeschaltet worden, führen sie zu Sprüngen und Unterroutinen, die mit dem Befehl ON COLLISION GOSUB festgelegt werden können.

ON COLLISION GOSUB label

Haben wir also das Kollisions-Trapping angeschaltet und mit ON COLLISION GOSUB umgeleitet, springt unser Programm automatisch auf die richtige Stelle (in diesem Fall zu »label«).

Mitunter kann es aber auch passieren, daß bei einigen Objekten eine Berührung miteinander vorkommen kann, diese aber nicht festgestellt werden soll. Mit dem Amiga ist es möglich, auszuwählen, Kollisionen welches Objekts mit welchem anderen Objekt wichtig sind und welche ignoriert werden können. Der Befehl dazu, OBJECT.HIT, ist allerdings nicht gerade leicht verständlich.

OBJECT.HIT Objektnummer ',MeMask' ',Hitmask'

Diese »Masken« sind Werte, deren binärer Wert dem Computer sagt, welche Objekte miteinander kollidieren dürfen. Die Masken »MeMask« und »HitMask« sind beide jeweils 16 Bit breit. MeMask beschreibt das Objekt, das durch den OBJECT.HIT-Befehl angewählt wurde, HitMask beschreibt das Objekt, mit dem es zusammenstößt (die Namen der Masken entstammen dem ROM-Kernel Manual und wurden auch ins BASIC übernommen). Der Amiga führt bei einer Berührung ein logisches UND zwischen der MeMask des einen Objektes und der HitMask des anderen Objektes durch. Dadurch erkennt der Computer, ob die Kollision stattgefunden hat oder ob sie übergangen wird. Ist das niederwertigste Bit in HitMask gesetzt, wird eine Kollision mit den Windowgrenzen registriert. Andernfalls wird eine Kollision mit den Windowgrenzen einfach übergangen. Die restlichen Bits sind für die Feststellung von Kollisionen zwischen den verschiedenen Objekten vorgesehen.

Das AmigaBASIC-Handbuch gibt allerdings nicht sonderlich viel Auskunft über diese Masken. Es verweist lediglich auf das ROM-Kernel-Manual. Daher scheint es sinnvoll, die Funktion dieser Masken etwas ausführlicher zu beschreiben. Nehmen wir als Beispiel drei Objekte aus einem Asteroids-Spiel: Objekt 1 soll ein

Asteroid sein (durch Kopieren mit dem OBJECT.SHAPE-Befehl erhalten wir mehrere davon), Objekt 2 ein Raumschiff und Objekt 3 ein Schuß.

Der Befehl OBJECT.HIT 1,8,7 beschreibt, daß Kollisionen eines Asteroids mit der Windowgrenze, dem Schiff und den Schüssen festgestellt werden können (Malen Sie sich zum besseren Verständnis hier die binären Werte der MeMask und Hitmask auf ein Blatt Papier).

OBJECT.HIT 2,2,9 sagt dem Computer, daß Kollisionen eines Schiffes mit der Windowgrenze und einem Asteroiden gemeldet werden soll.

OBJECT.HIT 3,4,9 bewirkt, daß Schüsse mit der Windowgrenze und den Asteroiden kollidieren können, daß eine Kollision zwischen Schüssen und einem Raumschiff jedoch keine Rolle spielt (Haben Sie sich schon mal selbst abgeschossen?).

In diesem Beispiel entsprechen die MeMasken der Objekte 1, 2 und 3 den Werten 8, 2 und 4. Das entspricht den Bits 3 ($2^3 = 8$), Bit 1 und Bit 2 in der jeweiligen Zahl. Die HitMask eines jeden Objektes beschreibt die Objekte, mit denen das entsprechende Objekt kollidieren kann. Bit 0 ist hier jedesmal gesetzt, da jedes der Objekte mit der Fenstergrenze kollidieren soll. Die restlichen Bits entsprechen wie in der MeMask dem jeweiligen Objekt.

Durch diese Technik kann man nun genau auswählen, welches Objekt mit welchem kollidieren darf. Die Asteroiden sollen zum Beispiel einfach aneinander vorbeifliegen. Wenn wir die MeMask des Asteroiden mit seiner HitMask durch ein logisches UND verknüpfen, kommt eine Null dabei heraus. Das bedeutet, daß Asteroiden nicht miteinander kollidieren können.

Wie schon gesagt, springt ON COLLISION sofort in eine Unterroutine, wenn eine Kollision stattfindet (und wenn das Event-Trapping mit COLLISION ON aktiviert ist). In dieser Unterroutine werden wir alles behandeln, was bei einer Kollision geschieht. Also entweder eine Explosion, einen Bewegungsstop oder anderes. Zuvor müssen wir allerdings wissen, welches Objekt mit welchem anderen kollidiert ist. Das können wir alles mit der Funktion COLLISION feststellen. Deren erste Form ist COLLISION(0).

COLLISION (0)

COLLISION (0) teilt uns die Nummer des Objektes mit, das zuletzt an einer Kollison beteiligt war. Haben wir das erst einmal herausgefunden, können wir mit der zweiten Form von COLLISION herausfinden, welches andere Objekt an der Kollision beteiligt war.

COLLISION (Objektnummer)

Bei Kollisionen mit den Windowgrenzen ergibt sich -1 für Kollisonen mit dem oberen, -2 für den linken, -3 für den unteren und -4 für den rechten Fensterrand. Eine weitere Form der COLLISON-Funktion, COLLISION(-1), ergibt die Nummer des Windows, in dem die Kollision stattfand.

Der BASIC-Interpreter merkt sich bis zu 16 Kollisionen in einer sogenannten »event queue«, wörtlich übersetzt »Ereignis-Schlange«. Wenn 16 Kollisionen stattgefunden haben, wird keine weitere mehr in die Liste aufgenommen. *COLLISION (Objektnummer)* nimmt jeweils die erste Zahl aus der Schlange und macht so Platz für die nächste Kollision. *COLLISION (0)* nimmt sinnvollerweise keine Informationen aus der Schlange, da wir für dieselbe Kollision die COLLISION-Funktion ja nochmal aufrufen müssen, um das andere Objekt herauszufinden. Das Schlangen-Prinzip ist in einer langsamen Interpretersprache wie BASIC notwendig, da ein Programm nicht immer unbedingt schnell genug auf Kollisionen reagieren kann. Damit wir keine der Kollisionen »verpassen«, hat man dem Programmierer diese Hilfe gegeben.

13.1.4 Räumlicher Eindruck

Bewegung macht jedes Programm schöner. Sie allein macht jedoch noch keinen »tiefen Eindruck«. Auch räumliche Grafik ist oft wünschenswert. Dazu muß ein Objekt »vor« dem anderen Objekt erscheinen, es also überdecken, um einen räumlichen Eindruck zu erzeugen. Auch da läßt uns Amiga BASIC nicht im Stich.

OBJECT.PRIORITY Objektnummer,Priorität

Der Befehl OBJECT.PRIORITY weist jedem Objekt eine Priorität zu. Der Wert kann jede Zahl zwischen -32768 und 32767 sein. Passieren zwei Objekte die gleiche Stelle des Bildschirms, überlagert das

Objekt mit der höheren Priorität das mit der niedrigeren. Diese Prioritätsbestimmung gilt jedoch nur für Bobs, denn Sprites haben eine automatische Priorität voreinander: Sprite 1 überlagert Sprite 2, Sprite 2 überlagert Sprite 3 und so weiter. Ein auf Sprites angewendeter Prioritätsbefehl bewirkt gar nichts.

Manchmal ist es aber auch störend, wenn ein BOB oder Sprite genau über den Text fliegt, den man gerade lesen will. Zu diesem Zweck gibt es das sogenannte »Clipping«, das durch den Befehl OBJECT.CLIP kontrolliert wird.

OBJECT.CLIP (x1,y1)-(x2,y2)

Durch OBJECT.CLIP werden alle Objekte, deren Position außerhalb des so angewählten Rechtecks liegen, automatisch unsichtbar – und wieder sichtbar, wenn sie in diesen Bereich zurückkehren.

13.1.5 Weitere Object-Befehle

Wenn wir unsere Objekte überhaupt nicht mehr brauchen, sollten wir natürlich den durch sie beanspruchten Speicher für das restliche Programm – oder für andere Animation – freigeben. Das erreichen wir mit dem Befehl OBJECT.CLOSE.

OBJECT.CLOSE Objektnummer.

Speicherplatzprobleme können uns die BOBs schon mal bereiten, wenn wir zuviele davon verwenden. Zu ganz speziellen Problemen bezüglich Farbe und Speicherplatz haben sich die Programmierer des AmigaBASIC den Befehl OBJECT.PLANES einfallen lassen.

OBJECT.PLANES Objektnummer ',PlanePick' ',PlaneOnOff'

Mit diesem Befehl können wir direkt auf zwei Parameter der Animationsbibliotheken (der GELS-Library innerhalb der Grafiklibrary; siehe auch die Einführung in das Betriebssystem später in diesem Buch) des Amiga-Betriebssystems zugreifen. Leider sind sie im BASIC-Handbuch nicht beschrieben, da diese Option für fortgeschrittene Programmierer gedacht ist. Deshalb hole ich hier das Versäumte nach. Bitte lesen Sie langsam und aufmerksam, denn so einfach ist die Sache nicht.

Wie Sie wissen, ist ein BOB ein Objekt, das per Software (die die Blitter-Hardware nutzt) erzeugt wird. Das heißt, das Objekt wird in die Bitplanes einkopiert, die auch die Hintergrundgrafik darstellen. Nun muß aber ein BOB nicht immer unbedingt genau so viele Farben haben wie der Hintergrund. Der Hintergrund kann zum Beispiel 32 Farben, also 5 Bitplanes haben, und das BOB braucht nur vier Farben. Wer faul ist und keine Speicherplatzprobleme hat, nimmt nun einfach den Objekteditor für 5 Bitplanes und benutzt nur einen Teil der verfügbaren Farben. Aber wozu 3 Bitplanes des BOBs verschwenden? Die Lösung des Problems liegt im OBJECT.PLANE-Befehl.

Wir passen also den Objekteditor auf zwei Bitplanes an und erstellen ein 2 Planes tiefes Objekt. Der OBJECT.PLANES-Befehl legt nun fest, wie die zwei Bitplanes in die 5 anderen Bitplanes kopiert werden sollen. Die zwei Parameter »PlanePick« und »PlaneOnOff« bestimmen, wie das geschehen soll. Der BASIC-Interpreter leitet diese zwei Parameter direkt in die Datenstrukturen des Betriebssystems weiter (Alle Grafikparameter werden in Datenstrukturen abgelegt, die das Betriebssystem bearbeiten kann). Sowohl PlanePick als auch PlaneOnOff können bis zu 8 Bits breit sein. Der Inhalt von PlanePick bestimmt, in welche Bitplanes des Hintergrundes die einzelnen BOB-Planes kopiert werden (also zum Beispiel BOB-Plane 1 in Hintergrund-Plane 2, Plane 2 in Hintergrundplane 5 und so weiter). Der Inhalt von PlaneOnOff bestimmt, was mit den Bitplanes des Hintergrundes passiert, in die die BOB-Planes nicht geschrieben werden.

Das sollten wir am besten mal an einem Beispiel durchgehen. Schließlich wollen wir nicht Bahnhof verstehen, sondern die Funktionsweise der beiden Parameter. Nehmen wir also an, wir haben ein 2 Bitplanes tiefes Blitter-Objekt definiert. Der Hintergrund, in den das Objekt kopiert werden soll, ist aber 5 Bitplanes tief. Wir wollen, daß die BOB-Farben aus den Farbregistern 24 bis 27 geholt werden. Um diesen Erfordernissen gerecht zu werden, müssen die Inhalte von Planepick und PlaneOnOff folgendermaßen sein:

```
PlanePick  00011  '(dezimal 3)
PlaneOnOff 11000  '(dezimal 24)
```

Die Bits der beiden Parameter stellen die Zuordnungen der einzelnen Bitplanes dar. Die Bits werden von rechts nach links gelesen und stellen jeweils eine Bitplane dar. PlanePick hat eine 1 in jeder

Position, in der eine Bobplane angezeigt werden soll. Die Bits werden folgendermaßen interpretiert.

Plane-Nr.	Bitinhalt	Bedeutung
1	1	1. BOBplane Hintergrundplane 1
2	1	2. BOBplane Hintergrundplane 2
3	0	Nicht selektiert
4	0	Nicht selektiert
5	0	Nicht selektiert
6	0	frei für zukünftige Erweiterungen
7	0	frei für zukünftige Erweiterungen
8	0	frei für zukünftige Erweiterungen

Tabelle 13.1: Die Bedeutung der PlanepickBits

Warum gehen wir hier bis Bitplane Nummer 8, wenn die Hardware nur 5 Bitplanes verträgt? Die Antwort ist einfach: Die Betriebssystemroutinen des Amiga sind bereits für zukünftige Erweiterungen ausgelegt, das heißt, wenn eines Tages eine Ausgabelogik entwickelt wird, die 8 Bitplanes verwalten kann, kann die alte Systemsoftware nach wie vor benutzt werden.

Laut obiger Tabelle wird also die erste Bitplane des BOBs auch in die erste Bitplane des Hintergrundes kopiert, wenn das BOB dargestellt wird, und die zweite BOB-Plane wird in die zweite Hintergrund-Bitplane kopiert. (Das geschieht nur dort, wo die Kombination der BOB-Planes nicht gerade Null ist, denn die Null bedeutet ja Transparenz.) In diesem Zusammenhang muß ich den – ebenfalls von den Amiga-Betriebssystem-Entwicklern geprägten – Begriff »Shadowmask« erwähnen. Die Shadowmask ist eine logische OR-Verknüpfung aller Bitplanes des BOBs und stellt somit die Form des BOBs in »Schwarzweiß« dar. Das sind also sind alle Punkte, die nicht die Hintergrundfarbe (Transparenz) tragen. Im Zusammenhang mit dem PlanePick-Parameter heißt das, daß in allen Stellen des Bildes, wo eine 1 in der Shadowmask sitzt, der Inhalt der BOB-Planes 1 und 2 in die Hintergrund-Bitplanes 1 und 2 geschrieben wird.

Der PlaneOnOff-Parameter besagt, was mit all den Hintergrundbitplanes passiert, die NICHT durch PlanePick angewählt sind. In unserem Beispiel tut er das für die Hintergrund-Bitplanes 3 bis 5. Die PlaneOnOff-Bits, die für die PlanePick-selektierten Planes gelten, werden einfach ignoriert.

Mit dem PlaneOnOff-Parameter können wir zum Beispiel die vorher erwähnten vielen Nullbytes, die die restlichen Planes füllen sollten, »simulieren«. Ist der PlaneOnOff-Parameter richtig gesetzt, werden beim Darstellen des BOBs die restlichen Planes automatisch mit diesen Nullen gefüllt, ohne daß wir in der Variable, die das BOB beschreibt, viele Nullbytes gespeichert haben müßten. Durch Ändern des PlanOnOff-Wertes könnten wir dann auch ein schnelles Umstellen der BOB-Farbe erzeugen, ohne daß wir alle Bitplanes des BOBs ändern müßten. Die Bedeutung der Bits wollen wir uns wieder anhand einer Tabelle betrachten.

Bit-Nr.	Bitinhalt	Bedeutung
8	0	für zukünftige Erweiterungen
7	0	für zukünftige Erweiterungen
6	0	für zukünftige Erweiterungen
5	1	Bits Plane 5 setzen
4	1	Bits Plane 4 setzen
3	0	Bits Plane 3 löschen
2	x	Inhalt egal
1	x	Inhalt egal

Tabelle 13.2: Die Bedeutung der Bits in PlaneOnOff

Für jedes gesetzte Bit wird die Shadowmask des BOBs benutzt, um die entsprechende Plane mit einer 1 zu füllen. Das heißt, überall dort, wo die Shadowmask des BOBs eine 1 enthält (also wo kein transparenter Punkt ist), werden die Hintergrundplanes, in die das BOB nicht durch PlanePick kopiert wird UND das PlaneOnOff-Bit 1 ist, mit 1 gefüllt. Und überall dort, wo die Shadowmask 1 ist, die Plane nicht durch PlanePick selektiert ist, und das PlaneOnOff-Bit 0 ist, wird die entsprechende Plane mit Nullen gefüllt.

Für unser Beispiel heißt das, Bitplane 5 und Bitplane 4 erhalten überall dort, wo das BOB nicht transparent ist, eine 1 verpaßt; Bitplane 3 wird dort mit Nullen gefüllt. Das bedeutet für die Farbwahl, daß die Farben 24 bis 26 für das BOB verwendet werden. Setzen wir jetzt im PlaneOnOff-Wert das Bit für die fünfte Bitplane auf 0, werden die Farben 8 bis 12 verwendet. Setzen wir jetzt auch noch das Bit für die vierte Bitplane auf 0, werden Farbregister 0 bis 4 für das BOB verwendet. Durch geschicktes Ändern des PlaneOnOff-Wertes kann man so (bei entsprechender Farbwahl) auch interessante Farbanimation erzeugen. Eine Anwendungsmöglichkeit wären zum Beispiel an- und aussgehende Lampen an einem Ufo. Für unser Beispiel müßte der entsprechende Befehl dann lauten:

```
OBJECT.PLANES Objektnummer,3,24
```

Für den letzten Wert ließe sich auch 25 oder 26 einsetzen, denn die Bits für Planes 1 und 2 werden ja bereits durch PlanePick reserviert und somit für PlaneOnOff uninteressant.

Wird kein OBJECT.PLANES-Befehl gegeben, aber ein Objekt verwendet, das weniger Bitplanes als der Hintergrund hat, so werden vom BASIC-Interpreter die BOB-Planes 1 bis 5 (oder weniger) in der selben Reihenfolge den Hintergrundplanes zugewiesen. Das heißt, wenn ein BOB drei Planes hat, wird BOB-Plane 1 in Hintergrundplane 1 gesetzt, BOB-Plane 2 in Hintergrundplane 2, und so weiter. Der Wert für PlaneOnOff liegt per Default-Wert bei 0, es werden also alle restlichen Planes mit Nullen gefüllt. Damit beginnt die Farbwahl immer bei Farbregister 0 (oder besser gesagt bei der Farbe 0, die für diesen SCREEN gilt. Die mit dem PALETTE-Befehl angewählten Farbwerte gelten immer nur für den Screen, der beim Farbbefehls-Aufruf aktiv war).

Damit sind wir nun endlich am Ende der Beschreibungen der Grafikobjekte Sprites und BOBs und haben bereits die ersten Animationstechniken erlernt: Bewegung per OBJECT-Befehle, Farbanimation von BOBs durch Ändern der PlaneOnOff-Werte. Wie Sie sehen können, bieten die Befehle für Sprites und BOBs dem BASIC-Programmierer eine ganze Menge des Animationspotentials des Amiga.

Aber auch in den normalen Grafikbefehlen wie LINE, PSET, CIRCLE oder PAINT steckt ein gewisses Animationspotential. Besonders schnell ist die Grafik gefüllter Polygone mit den Befehlen AREA und

AREAFILL. Auch die PUT-, GET- und SCROLL-Befehle leisten eine ganze Menge. Der nächste große Abschnitt dieses Buches wird Ihnen zeigen, wie Sie in der BASIC-Praxis zu besseren Animationen kommen.

13.2 Animationstechniken

Die Animationsfähigkeiten, die in den einzelnen BASIC-Befehlen versteckt sind, brauchen wir eigentlich nur noch aus AmigaBASIC herauszukitzeln.

13.2.1 Bewegung mit Scrolling

Beginnen wir zuerst mit dem Scrolling: »Scrolling« ist der Fachausdruck für die Bewegung einer Hintergrundgrafik (nicht die Bewegung von Objekten!). Der Name des zugehörigen Befehls, SCROLL, bezieht sich auf dieses Fachwort. Fangen wir gleich mit einem kleinen Beispiel an:

```
REM Scrolldemo
 WINDOW 1,"Scrolldemo"  'Verwende Basic-Output-Window
 WINDOW OUTPUT 1        'Alle Ausgaben in dieses Window
 MOUSE ON               'Mausabfrage einschalten
   loop:
     painter            'Gehe in Unterroutine painter
     scroller           'Gehe in Unterroutine scroller
   GOTO loop:
 SUB scroller STATIC
 SCROLL (10,10)-(250,250),1,2 'Scrolling
 END SUB
 SUB painter STATIC
   x=MOUSE(1)
   y=MOUSE(2)
   button=MOUSE(0)
   IF button <> 0 THEN PSET(x,y)  'Wenn Mausbutton unten
                                  'dann Punkt setzen
 END SUB
```

Dieses kurze Listing zeigt ganz deutlich, wie der SCROLL-Befehl zu handhaben ist. Er verschiebt jedesmal, wenn er aufgerufen wird, nur

so viele Pixels, wie im Befehl angegeben sind. Schöner wäre es natürlich gewesen, hätten uns die Programmierer des BASIC eine Möglichkeit gegeben, den Hintergrund unabhängig vom Rest des Programmes zu verschieben (so wie bei den Objekten). Leider taten sie das nicht.

Wie setzen wir diese Kenntnis nun für professionelle Animation, beispielsweise für Geschäftsgrafiken oder Spiele ein? Eine Möglichkeit wäre, die bereits gezeichnete Balkengrafik langsam nach oben verschwinden zu lassen, um so effektvoll Platz für die nächste Grafik zu machen. Das gleiche kann man natürlich auch mit einem Titelbild eines Spieles tun. Oder man zieht das Bild in die Mitte des Bildschirms zusammen, so wie es im AmigaTutor recht eindrucksvoll erscheint. Das läßt sich sogar schon mit einem außerordentlich kurzen Programm erreichen. Das folgende Listing enthält ein solches Programm für ein 640 mal 200 Punkte großes Bild (also den Standard-Workbench-Screen):

```
REM Hier käme erst die Routine für Bildaufbau
REM Und jetzt das Zusammenziehen des Bildes:
scrolling:
    SCROLL (1,1)-(300,90),2,3
    SCROLL (360,1)-(600,90),-2,3
    SCROLL (1,91)-(300,180),2,-3
    SCROLL (300,91)-(600,180),-2,-3
  x=x+1
  IF x < 31 THEN
   GOTO scrolling
  ELSE
   LOCATE 12,35
   PRINT "Peng!"
  END IF
```

Das Ergebnis sieht aus, als ob die Bildschirmmitte das restliche Bild in sich einsaugt. Wenn Sie das allerdings so eintippen, wie es jetzt ist, kommt nichts dabei heraus, denn Sie haben ja noch kein Bild auf dem Bildschirm. Leider »ruckelt« es furchtbar, aber das liegt an der Geschwindigkeit von BASIC, die uns allen leider das Leben schwer macht.

Wie Sie wissen, verschwindet alles, was aus dem Bereich des Scrolling herausbewegt wird, im Nichts. Wollen wir eine fort-

laufende Grafik erstellen, müssen wir also in der Richtung, aus der das Scrolling kommt, für Nachschub der Grafik sorgen. Bei einem Defender-Spielchen müßte – bei Rechtsbewegung – also immer ganz links ein neuer Punkt gesetzt werden. Die Reihenfolge Punkt setzen – scrollen – Punkt setzen – scrollen und so weiter ergibt zusammen dann die gewünschte Gebirgskette, die vorbeirauschen soll. Gehen wir diese Möglichkeit wieder an einem Beispiel durch. Das folgende Beispielprogramm zeigt eine Strecke, auf der wir später ein Auto fahren lassen wollen. Die Strecke soll von oben nach unten scrollen und so die Straße darstellen, auf der wir entlangfahren.

```
SCREEN 1,320,200,2,1     'Bildschirm in 4 Farben, LoRes
WINDOW 1,"Autobahn",,,1 'Window 1 in voller Größe
                         'auf Screen 1
WINDOW OUTPUT 1          'Ausgaben alle in Window 1
PALETTE 0,0,0,0          'Hintergrundfarbe schwarz
PALETTE 1,.4,.3,.2       'Window- und Textfarbe braun
PALETTE 2,.2,.7,.1       'Ekliger Grünton für Autobahn

start:
 x%=(RND*2)           'Zufallszahl für links oder rechts
 IF x%=0 THEN z%=1    'Wenn 0, dann x-Position+1 (rechts)
 IF x%=1 THEN z%=-1   'Wenn 1, dann x-Position-1 (links)
 IF x%=2 THEN z%=0    'Wenn 2, dann geradeaus
 IF y%<20 THEN z%=1   'Wenn zuweit links, nach rechts
 IF y%>90 THEN z%=-1  'Wenn zuweit rechts, nach links
  FOR i=1 TO 5        '5 = Größe der Kurven
  y%=y%+z%            'x-Position = alte Pos. + Richtung
  LINE (y%,11)-(y%+15,11),2    'Linie für Straße, 15
                               'Pixel breit, in Farbe 2.
 SCROLL (10,10)-(100,200),0,1 'Bewegung nach unten
  NEXT
GOTO start
```

Und schon bewegt sich unsere Straße. Erhöhen wir den Wert am Ende der FOR-Schleife, sind die Kurven größer und damit einfacher befahrbar. Nehmen wir die Zeile für die Geradeaus-Bewegung heraus, ist die Strecke kurvenreicher. Die Breite der Straße können wir durch den zweiten x-Wert des LINE-Befehls erhöhen. Durch Einsetzen von Variablen in diese Programmzeilen können wir die verschiedenen Schwierigkeitsgrade für ein Spiel variabel halten.

Den SCROLL-Befehl haben wir nur ein einziges Mal in diesem Listing stehen, und trotzdem dreht sich alles nur um das Scrolling. Sie sehen, es ist nicht nur der Befehl, der etwas macht, sondern es ist das Drumherum. Pro Kurve wird der SCROLL-Befehl fünfmal aufgerufen, und erst diese Verwendung innerhalb einer großen Schleife macht den Gesamteindruck perfekt.

Sicher wird Ihnen an dem Listing noch etwas aufgefallen sein: Die Prozentzeichen nach den Variablennamen. Das bedeutet, daß wir mit Integer-Variablen arbeiten. Damit kann das BASIC schneller arbeiten als mit den Fließkommazahlen einfacher oder doppelter Genauigkeit. Geben wir hinter einem Variablennamen nichts an, nimmt das BASIC nämlich Zahlen einfacher Genauigkeit an. Bei einem so kurzen, kleinen Programm ist der Geschwindigkeitsvorteil noch unmerklich; doch bei größeren komplizierteren Problemen kommt die Zeiteinsparung durch Integer-Variablen bereits zum Tragen. Doch auch an anderen Stellen ist unser Demoprogramm hier noch nicht vollkommen geschwindigkeitsoptimiert. Der BASIC-Interpreter geht ja bekanntlich das Listing Byte für Byte durch, erkennt die Befehle und führt sie dann erst aus. Geben wir jedesmal das Prozentzeichen an, um eine Integervariable zu definieren, kostet das natürlich Zeit. AmigaBASIC erlaubt uns, wie in einem Pascal-Programm, die Variablen vorher als Integer zu definieren. So muß nicht erst der BASIC-Interpreter suchen, ob er es mit einer Integer-Variable zu tun hat. Trifft er beim Durchgehen des Listings auf den Variablennamen, weiß er automatisch, wie er diese Variable zu bearbeiten hat. Setzen wir also am Anfang des Listings den folgenden Befehl.

DEFINT x,y,z,i

Dann sind x,y,z und i als Integer-Variablen deklariert. Wir können dann die Prozentzeichen an den entsprechenden Variablen entfernen und diese werden trotzdem als Integer-Variablen verarbeitet. Damit haben wir sowohl Zeit als auch Speicherplatz gespart.

Experimentieren Sie nun ruhig etwas mit dem Listing herum. Setzen Sie die Breite der Straße auf 30, die Größe der Kurven auf 10. Das Scrolling ist immer noch schnell, aber je größer Sie die Breite der Straße setzen, desto langsamer wird die Grafik. Daraus resultiert eine Tatsache, die Sie sich merken müssen, wenn Sie Animation pro-

grammieren wollen: Je größer die Grafik, die wir animieren, desto langsamer die Geschwindigkeit des Programms.

Setzen wir nun einmal den Wert für die Anzahl der Pixels, die gescrollt werden soll, um 1 hoch, also:

SCROLL (10,10)-(100,200),0,2

Nach dem Programmstart erkennen wir zunächst eine eindeutige Geschwindigkeitssteigerung. Das ist auch ganz klar, denn wir bewegen ja dieselbe Grafikmenge in einer Bewegung um das Doppelte. Dafür aber haben wir in der Grafik der Straße Leerzeilen. Daraus resultiert unser nächster Merksatz für die Animationsprogrammierung: Je schneller die Grafik sein soll, desto weniger detailreich kann sie sein.

Die Leerzeichen können wir jetzt wieder ausgleichen, indem wir hinter den ersten LINE-Befehl einen zweiten setzen:

LINE (y,12)-(y+30,12),2

Bitte beachten Sie, daß diese zusätzliche Zeile für das veränderte Listing mit Straßenbreite 30 und Kurvenparameter 10 gilt. Starten Sie nun. Ergebnis: Durch Zeichnen von zwei Linien untereinander haben wir das Problem der Leerzeilen gelöst und die Geschwindigkeit ist größer als das vorherige Programm mit SCROLL-Geschwindigkeit 1. Dafür aber ruckt und zuckt es etwas stärker als vorher. Das fällt zwar noch kaum ins Gewicht, doch je mehr wir diese Methode ausbauen, desto flimmernder wird das Bild. Beispielsweise bei Scrollgeschwindigkeit 8 und 8 LINE-Befehlen flimmert es am oberen Rand der Straße schon ziemlich stark. Daraus folgt unser dritter Merksatz für die Animation: Soll eine Animation sowohl schnell als auch detailreich sein, leidet der optische Eindruck darunter.

An einer Änderung auf so viele LINE-Befehle ist aber noch ein anderer Haken: Unsere Straße ändert sich, sie ist nicht mehr so kurvenreich. Kein Wunder, denn für jede Phase der Rechts- oder Linksbewegung haben wir ersteinmal 8 gerade untereinanderliegende Linien. Um die Straße wieder kurvenreicher zu machen, müßten wir also den »Straßenerzeuger-Algorithmus« ändern. Und neue Berechnungen brauchen wieder mehr Rechenzeit.

Sollte aber diese Straße sie zufriedenstellen, können wir uns nun etwas neues überlegen. Denn ein so zeitkritisches Unterfangen wie Animation ist in erster Linie eine Frage der Optimierung. Wir müssen selbst erkennen können, was die schnellsten Methoden sind. Sehen Sie sich das Listing an. Wo könnte man etwas besser machen? Überlegen Sie erst einmal, bevor Sie den nächsten Absatz lesen, denn wenn Sie selbst programmieren, haben Sie ja auch niemanden, der Ihnen sagt, wo es langgeht.

Sie haben wirklich keine Ahnung? Nun, zum einen könnte das vielleicht der Algorithmus zur Berechnung einer Grafik sein. Er ist es aber in diesem Fall nicht. Zumindest mir ist nichts Besseres oder Kürzeres eingefallen. Aber vielleicht beim Zeichnen der Grafik selbst? Richtig, wir haben ja so viele LINE-Befehle hintereinander, die noch dazu alle untereinander liegende Linien ziehen. Tja, wenn das kein Rechteck ist?

Sie sehen, worauf ich hinaus will: Statt der vielen Linien brauchen wir nur ein einziges Rechteck zu malen, also genügt ein einziger LINE-Befehl mit dem Anhang »,bf«. Gehen wir wieder auf unser Listing zurück, das wir einmal mit dem Scrollparameter 5 versehen wollen. Die Grafik geht also jedesmal um 5 Pixels nach unten, wenn der Scroll-Befehl aufgerufen wird. Also müssen wir auch ein 5 Pixels langes Rechteck zeichnen, damit keine Lücken in der Straße auftreten. Der Befehl dazu ist dann also:

LINE (y,11)-(y+30,15),2,bf

Dadurch, daß wir ein Rechteck von y-Position 11 bis 15 zeichnen, vermeiden wir die vier LINE-Befehle, die jede Zeile von 12 bis 15 gezeichnet hätten. Durch diese Optimierung muß der BASIC-Interpreter nur noch eine einzige Zeile interpretieren, was natürlich ein Geschwindigkeitsvorteil ist. Merksatz Nummer 4: Optimiere, wo es geht – Möglichkeiten gibt es immer dafür.

Nun können wir natürlich noch nach weiteren Möglichkeiten suchen, die Grafik zu verbessern. Geschwindigkeitsmäßig haben wir bereits optimiert. Wie erreichen wir nun noch eine Verbesserung des optischen Eindrucks, ohne die Geschwindigkeit wesentlich zu verlangsamen? Die Lösung liegt dabei immer in der Art der Grafik verborgen. In unserem Beispiel handelt es sich um rechteckige Flächen. Wie können wir rechteckige Flächen noch darstellen und gleichzeitig mehr Details in die Grafik bringen? Antwort: Der PUT-Befehl

liefert uns rechteckige Flächen. Und: Mit einem PUT-Befehl können wir beliebige detailreiche rechteckige Grafiken aus einem mit GET geholten Feld in die Grafik setzen.

Wir brauchen also nur die einzelnen Teile einer Straße zu malen, mit dem GET-Befehl aus der Grafik zu holen und das Variablenfeld entweder für die spätere Benutzung auf Diskette abspeichern oder in Data-Zeilen abzulegen.

Das Ablegen auf Diskette ist speicherplatzsparender, denn hier brauchen wir keinen Speicherplatz für DATA-Befehle. Die DATA-Methode erlaubt es, auch ohne Nachladen von Diskette klarzukommen. Mir persönlich ist die Diskettenmethode lieber. Die Entscheidung, was für Sie am ehesten geeignet ist, überlasse ich Ihnen.

Für unsere Straße benötigen wir eigentlich nur drei verschiedene Grafiken: Einmal die nach links abbiegende Straße, einmal die nach rechts, und einmal das gerade Zwischenstück. Je nach Straßenlage (in unserem Listing der Wert z) plazieren wir dann die entsprechenden Teile in die Grafik.

Wir können also durchaus auch detailreiche Grafiken schnell animieren. Aber nur, solange der Detailreichtum unabhängig vom Befehl ist, mit dem gezeichnet wird. Dem PUT-Befehl ist es grundsätzlich egal, wieviele Details unsere Grafik hat. Er speichert einfach nur den Inhalt eines Variablenfeldes in die Grafik, egal wie detailreich die Grafik in diesem Variablenfeld ist. Gleiches passiert übrigens auch mit den Sprites. Ganz gleich, ob Sie nur einen Strich als Sprite genommen haben oder ein Gesicht, die Bewegungsgeschwindigkeit ist die gleiche. Anders sieht es da schon bei Grafiken aus, die in Ihrem Detailreichtum von den Befehlen abhängig sind. Zeichnen wir also Grafiken mit dem LINE-Befehl, hängt der Detailreichtum natürlich von der Anzahl der LINE-Befehle ab. Und bei steigender Anzahl der Befehle sinkt auch die Geschwindigkeit.

Aus den Kenntnissen, die Sie bisher aus dem Scrollen gezogen haben, wird folgendes klar: Animation ist eine Sache, die nur dann gut wird und glatt läuft, wenn alle Komponenten gut aufeinander abgestimmt sind. Die Komponenten guter Animation sind:

- Algorithmen und Berechnungen, die das Aussehen der Grafik bestimmen.

- Die Anzahl der Befehle, mit denen die Grafik gezeichnet wird.
- Die Parameter der Grafikbefehle
- Der Detailreichtum der Grafik
- Spezielle Techniken bei der Anwendung der Grafikbefehle
- Speicherplatzverbrauch

Das Ziel bei der Programmierung von Animation ist es, die optimale Kombination aller dieser Komponenten zu finden. Jede einzelne Komponente muß bereits optimiert sein, um in der Gesamtheit dann nochmals optimiert zu werden. Manchmal muß man eine Komponente zugunsten einer anderen vernachlässigen, um einen besseren optischen Effekt hervorrufen zu können. Was man vernachlässigen kann beziehungsweise was man verbessern muß, kann ich Ihnen nicht pauschal darlegen. Denn die Optimierung hängt immer vom jeweiligen Problem ab.

Um unser Scrolling noch weiter zu optimieren, gehen wir die Sache einmal von einem anderen Blickwinkel an. Wir wollen möglichst detaillierte Grafik möglichst stufenlos scrollen. Ist es dazu unbedingt nötig, bei jedem Scroll-Schritt auch die Grafik zu setzen? Wir können doch auch immer 20mal um jeweils ein Pixel scrollen, dann den nächsten Teil der Grafik in die frei gewordenen Stellen setzen, und dann erst weiterscrollen. So müssen wir nicht bei jedem Scroll-Schritt die Grafik ändern und sparen uns viel Zeit. Gehen wir zurück auf das Defender-Spiel, wo die Berge horizontal scrollen müssen. Sehen Sie sich dazu folgendes Beispiel an.

```
Groesse=20
 WHILE 1
  FOR i = 1 TO Groesse:GOSUB scrolling:NEXT
  GOSUB setzen
 WEND

scrolling:
SCROLL (1,10)-(630,500),1,0
RETURN
```

```
setzen:
 z=z+1
 IF z/2=INT(z/2) THEN
  LINE (1,10)-(Groesse,50)
 ELSE
  LINE (1,50)-(Groesse,10)
 END IF
RETURN
```

Wie Sie sehen, scrollen wir immer 20mal, bevor wir den neuen Bildteil setzen. Die Bedingung (z/2=INT(z/2)) ist jedes zweite Mal erfüllt, das heißt einmal geht ein Strich nach oben, einmal geht ein Strich nach unten. Wird der Parameter Größe verändert, so werden die Linien flacher; insgesamt ergibt sich ein (minimaler) Geschwindigkeitsvorteil, dafür aber sieht man um so besser, wie links die Grafik nur ruckhaft erscheint.

Der Vorteil dieser Technik ist, daß das Scrolling stufenlos ist und wir trotzdem nicht jedesmal einen Teil der Grafik setzen brauchen. Den Nachteil, nämlich das ruckhafte Erscheinen, können wir umgehen. Wir setzen einfach etwas über den Grafikbereich, an dem das Bild erscheint, ein Sprite oder ein Window (kein BOB! Das in die Grafik gesetzte BOB würde mitscrollen anstatt überdecken, da BOBs ja in die Grafik hineinkopiert werden). Wir nehmen dazu folgende Änderungen am Programm vor:

```
Groesse=20
WINDOW 1 'Basic-Output-Window
WINDOW 2,,(0,10)-(Groesse,50),0
WINDOW OUTPUT 1
 ' Und hier geht's wie vorher mit WHILE... weiter.
```

Dabei ist zu beachten, daß Window 1 ein Smart-Refresh-Window sein muß (Window-Typ ≤ 16), damit der vom Window überdeckte Teil auch gescrollt werden kann. Da das BASIC-Output-Window automatisch auf Smart-Refresh geschaltet ist, brauchen wir es aber nicht extra einzustellen. (Hätten wir mit einem OBJECT-Befehl ein SPRITE darübergelegt, hätten wir kein Smart-Refresh-Window nötig.)

Ist Window 1 gesetzt, setzen wir Window 2 über die Stelle, an der es ruckelt, danach schalten wir aber die Ausgabe auf Window 1 zurück,

damit das Scrolling nicht dort stattfindet, wo es eigentlich verdeckt werden soll.

Starten Sie das Programm. Stufenlos, ohne Rucken geht die Verschiebung nun vor sich. Aber vielleicht ist es etwas zu langsam. Na, da können wir ja wieder die Scroll-Geschwindigkeit anpassen. Wir führen die Variable gs% für die Geschwindigkeit ein. Diese Variable wird in den x-Scrollwert des SCROLL-Befehles gesetzt. Wenn wir jetzt starten, sieht das Bild nicht gerade schön aus. Wir müssen also auch noch das verdeckende Window und die LINE-Befehle anpassen, denn gescrollt wird ja jetzt doppelt so viel. Hinter alle »Größe«-Parameter in WINDOW- oder LINE-Befehle setzen wir dann den Anhang **gs%*, also zum Beispiel *LINE (1,10)-(Groesse*gs%,50)* für den ersten LINE-Befehl. Damit haben wir die Geschwindigkeit erhöht. Für noch schnelleres Scrolling muß nur noch der gs%-Wert geändert werden. Damit das überdeckende Window nicht ganz nutzlos ist, können wir es als grafisches Element benutzen, also zum Beispiel zur Darstellung des Radars oder irgendwelcher Raumschiff-Grafik. Diese Verschönerung überlasse ich Ihnen.

Wir sprachen von detailreicher Grafik. Hierzu können wir, wie vorher bereits vorgeschlagen, Grafik mit dem PUT-Befehl hineinsetzen. Anstatt nur Linien zu sehen, können wir jetzt also echte Berge in die Grafik setzen. Sie müssen nur vorher die Berge malen, sie dann mit GET in ein Array zwischenspeichern, welches Sie dann im Programm benutzen. Das Array kann mit einer kurzen Routine, OPEN und PRINT#, auf Diskette zwischengespeichert werden. Sehen Sie sich dazu bitte im Kapitel »Beispiele« die entsprechende Routine im IFF-Ladeprogramm an. Sie können so die Berge sogar mit Deluxe-Paint zeichnen, sie mit dem IFF-Ladeprogramm in BASIC einladen, um sich dann die entsprechenden Teile mit GET-Befehlen zu holen und sie auf Diskette zwischenzuspeichern. Die so zwischengespeicherten Berge können Sie dann im obigen Listing einsetzen. Sie müssen dabei darauf achten, daß die Berge links und rechts auf derselben Höhe aufhören, damit sie beim Nebeneinandersetzen zusammenpassen.

Auf das Scrolling mit Bildern werden wir später noch einmal zurückkommen, denn hier stecken enorme Fähigkeiten in einer Kombination der verschiedenen Möglichkeiten, die wir erst einmal miteinander erlernen werden. Unser endgültiges Ziel sind ja scrollende Grafiken hervorragender Bildqualität mit Farbanimation im Hintergrund und

sich bewegende Objekte, wie BOBs und Sprites, davor. Diese Grafik mag dann durchaus noch durch Windows unterstützt werden, in denen sich dreidimensionale Objekte drehen... Nun, alles das geht in BASIC aus Geschwindigkeitsgründen nicht gleichzeitig. Aber wir werden uns bemühen, so viel wie möglich davon zu realisieren.

13.2.2 Animation mit PUT und GET

Wenn Sie sich beim Kauf Ihres Amiga auch gleich die BASICdemos angesehen haben, kennen Sie ja sicher schon das Demo »Picture«. Zuerst wird ein Bild mit Hilfe des CIRCLE-Befehls aufgebaut, das dann mit GET in ein Array eingelesen wird. Je nach Position der Maus wird das Bild nun mit PUT in die Hintergrundgrafik einkopiert. Durch die Aktionsverben zur Darstellung der Grafik verbirgt sich hinter dem PUT-Befehl ein gewisses Animationspotential. Das will ich Ihnen anhand einiger kleiner Änderungen des Demos beweisen. Setzen Sie einmal vor das Hauptprogramm »Picture« die folgenden Zeilen.

```
painter:
 IF INKEY$=" " THEN main
 IF MOUSE(0)<>0 THEN PSET(MOUSE(1),MOUSE(2))
 GOTO painter

main:
 'Hier geht das Programm weiter wie vorher
```

Nehmen Sie nun noch den CLS-Befehl (steht nach dem DIM) heraus und starten Sie das Programm. Der erste Teil ist ein Minimal-Malprogramm, das ich immer verwende, wenn ich zur Überprüfung einer Sache einen Hintergrund brauche. Malen Sie damit nach Herzenslust herum. Drücken Sie dann die Space-Taste. Jetzt wird die Grafik eingelesen, die Sie dann mit der Maus bewegen können.

Wenn Sie das Bild bewegen, verschwindet der Hintergrund nicht. Und das aus dem einfachen Grund, daß wir unsere Grafik mit einer XOR-Verknüpfung in den Hintergrund setzen, danach den PUT-Befehl nochmals aufrufen und durch die nochmalige XOR-Verknüpfung wieder der ursprüngliche Inhalt im Bild steht. Damit können wir quasi ein rechteckiges BOB simulieren, ohne uns mit den vielen OBJECT-Befehlen herumzuquälen. Dafür aber gibt es keine Kollisionsfeststellung und keine von selbst laufende Bewegung. Für Zwecke, bei

denen einfach nur ein kurzes vorbeilaufendes Objekt benötigt wird, ist diese Methode aber ideal.

Man hat dabei zwar viel weniger Programmieraufwand und bewirkt das gleiche wie mit einem BOB, dafür aber reicht der Speicher nicht immer (Sie können das eventuell mit dem CLEAR-Befehl anpassen). Arrays benötigen in der Regel mehr Speicherplatz als eine Stringvariable, in der zum Beispiel ja auch BOBs gespeichert werden. Sie müssen anhand Ihres freien Speicherplatzes und der Komplexität Ihres Problems selbst entscheiden, ob Sie ein BOB oder ein PUT-Rechteck verwenden.

Statt als unabhängiges bewegtes Objekt können wir unser PUT-Array auch für andere Zwecke verwenden; zum Beispiel als Radiergummi. Ändern Sie den zweiten PUT-Befehl im Listing einmal, indem Sie hinten ein »,PSET« anhängen. Starten Sie nun das Programm erneut und sehen Sie, was mit dem Hintergrund geschieht. Überall, wo Sie nun mit der Maus hinklicken, wird sofort der Hintergrund gelöscht. Es sind neben der Radiergummifunktion auch noch andere Möglichkeiten denkbar: Durch andere Kombinationen der verschiedenen Aktionsverben kann man Pinsel in verschiedenen Farben simulieren und so weiter. Probieren Sie ruhig etwas herum, und Sie werden sicherlich viele Verwendungen für eigene Programme finden. Ein Beispiel dazu will ich Ihnen noch zeigen. Löschen Sie dazu alle Programmzeilen nach »Checkmouse:« und fügen Sie stattdessen folgende Zeilen an.

```
FOR x=600 TO 80 STEP -8
y=x/3
PUT(x,y),p,PRESET
PUT(x,y),p,PSET
NEXT x
END
```

Starten Sie jetzt das Programm. Sie sehen, das »Objekt« bewegt sich schnell über den Bildschirm und schleppt dabei eine Spur hinter sich her. Die Form der Spur kann man durch Ändern des STEP-Wertes in der Schleife und durch Ändern der Aktionsverben der PUT-Befehle verändern. Ist der Wert bei STEP -1 und beide Aktionsverben sind XOR, so bewegt sich das Objekt »majestätisch« über die Hintergrundgrafik weg, ohne sie zu zerstören. Auch hier gilt wieder: Experimentieren! Mit Hilfe des PUT-Befehls kann also auch eine be-

trächtliche Vielfalt von Animationen durchgeführt werden. So können wir auch ganze HiRes-Bilder bewegen, auf- und abscrollen und so weiter.

Das IFF-Programm im Beispiele-Kapitel verfügt über eine SAVE-Option für Bilder. Das Bild wird genauso abgespeichert, daß man es beim Wiedereinlesen als PUT-Array verwenden kann. Auch eine kurze Laderoutine dafür ist im Beispiele-Kapitel zu finden. Haben wir das Bild geladen, können wir es in den Bildschirm hereinscrollen, indem wir mit dem PUT-Befehl ganz einfach die Anfangskoordinaten ändern. Dazu schreiben wir eine kleine Unterroutine, die ein Scrolling in beliebige Richtungen mit dem PUT-Befehl realisiert.

```
Arraymove:
REM Parameter (x1,y1,x2,y2,speed)
dx=ABS(x2-x1):dy=ABS(y2-y1)           'Um wieviel Punkte
                                      'ändern sich x und y?
IF dx>dy THEN steps=dx ELSE steps=dy 'Anzahl Schritte
xs=dx/steps:IF x2<x1 THEN xs=-xs      'x-Änd. pro Schritt
x=x1+.5:y=y1+.5                       'Vermeiden von Rund-
                                      'dungsfehlern
FOR u=1 TO steps STEP speed
  CLS
  PUT (x,y),pic%,PSET        'Grafik setzen
  FOR z=1 TO speed
    x=x+xs:y=y+ys            'Werte+Schritt=neue Position
  NEXT
NEXT
CLS:PUT(x2,y2),pic%,PSET      'durch speed verursachte
                             Fehlberechnungen ausgleichen
RETURN
```

Wenn Sie sich das Listing einmal näher ansehen, merken Sie, woher wir die Berechnung der Bewegung genommen haben: Vom Line-Algorithmus, den wir im ersten Buchteil entwickelt haben und vergleichsweise auch im BASIC-Kapitel über Linien zeigten. Dadurch muß derjenige, der diese Routine benutzt, keine Werte mehr für die Zwischenschritte der Bewegung ausrechnen. Die Anfangs- und Endpunkte genügen, denn daraus errechnet sich die gerade Linie, auf der wir vom Anfang bis zum Ende unser Objekt bewegen.

Sie sehen also, daß der mathematische »Firlefanz« doch recht nützlich ist, auch wenn wir nicht selbst unsere Linien ziehen müssen. Sie müssen lernen, zur Animation auf dem Computer auch Mathematik einzusetzen. Das letzte Listing ist gut mit REMs ausgestattet und zeigt Ihnen genau, wie alles funktioniert. Für kreiselnde Bewegung (beispielsweise wie bei den Aliens im Automatenspiel »Galaxians«) können Sinus- und Cosinus-Funktionen verwendet werden. Und auch die ach so schönen Lissajous-Figuren mißbrauchen wir kurzerhand für die Formationen unserer UFOs.

Doch zurück zu unserem eigentlichen Problem: Die Routine Arraymove wird im Programm immer mit den beiden folgenden Zeilen aufgerufen.

```
x1=..:y1=..:x2=..:y2=..:speed=.. '(entsprechende Werte)
GOSUB Arraymove
```

Die Koordinaten (x1/y1) setzen den Startpunkt fest, (x2/y2) den Endpunkt. Gescrollt werden kann in jede beliebige Richtung. Der Geschwindigkeitsparameter »speed« ist die Anzahl der Pixels, um die sich das Bild bei jedem Schritt bewegt.

Mit dieser Routine sind wir auch bereits wieder beim Scrollen angekommen, der kleinen Marotte eines jeden, der etwas Bewegtes programmieren will (Wie wir den PUT-Befehl und den SCROLL-Befehl sinnvoll kombinieren, wird in einem späteren Kapitel beschrieben). Eine andere Marotte eines jeden »Animateurs« ist der Wunsch, bewegte dreidimensionale Grafiken zu erzeugen. Dazu brauchen wir aber nicht unbedingt gleich umständliche Berechnungen. Wir malen uns einfach die verschiedenen Bewegungsphasen des Objektes auf den Bildschirm und speichern sie mit GET entweder in verschiedene Variablenfelder, oder gleich in doppelt indizierte, weil dann die Behandlung des Bildes in Schleifen einfacher ist.

Das BASIC-Handbuch sagt zwar, daß das möglich ist, zeigt aber nicht genau, wie. Die Lösung dieses Problems fassen wir mit einer ganz kurzen Erklärung zusammen: Der GET-Befehl speichert im Normalfall die Grafik am Anfang des Variablenfeldes ab, bei einem eindimensionalen Array also bei Variable(0), bei einem zweidimensionalen Feld mit Variable(0,0) und so weiter. Geben wir aber zusammen mit dem GET-Befehl Indizes an, wird die Grafik ab den Elementen der angegebenen Indizes abgespeichert. Die ersten drei abgespeicherten Werte sind x-Breite, y-Höhe und Anzahl der Bit-

planes, danach kommt die Grafik. Genauso funktioniert der PUT-Befehl. Wird nichts angegeben, wird eine Grafik ausgegeben, die bei dem nullten Element des Feldes beginnt. Geben wir den Index eines Feldelements an, wird die Grafik ausgegeben, die ab diesem Element gespeichert ist.

Damit ist auch schon klar, wie wir unsere Animation programmieren: Mit dem DIM-Befehl legen wir zuerst den benötigten Speicherplatz fest. Für eine Animation mit 5 Bildphasen dimensionieren wir unser Feld also mit der folgenden Anweisung:

```
DIM Variable%(Groesse,5)
```

Dabei gibt »Groesse« die Feldgröße an, wie wir sie auch bei eindimensionalen Feldern berechnen. (Die Berechnungsroutine für die Größe des Feldes wird bei der Erläuterung des GET-Befehls beschrieben.) Der zweite Wert (5) ist die Anzahl der Bilder, die unser Film braucht.

Natürlich brauchen wir dazu – je nach Anzahl der Bilder – Unmengen von Speicher. Deshalb muß der Speicher vorher an den benötigten Speicherplatz angepaßt werden. Leider ist nicht immer so viel frei, wie man gerne möchte. Im Notfall müssen alle anderen Windows geschlossen werden (auch das LIST-Window). Manchmal ist es auch ratsam, das Output-Window kleiner zu machen. In dem folgenden Beispiel brauchen wir nur die linke Hälfte des Bildschirms, also können wir das Output-Window nach links ziehen. Sollte das alles keine Früchte tragen und der Speicher noch immer nicht reichen, um die gewünschte Bildanzahl darzustellen, muß man gleich beim Starten des Amiga mit der Abhilfe beginnen. Drücken Sie beim Starten der Workbench-Diskette mit CTRL–D (bevor loadwb ausgeführt wurde). Ziehen Sie dann das CLI-Window kleiner und starten Sie AmigaBASIC vom CLI aus. Das Beste, was ich auf diese Weise herausgeholt habe, waren 20 Bilder in dem folgenden Programm (mit einem Amiga 1000 mit 512 KByte RAM).

```
i=-1
INPUT "Animationsphasen  ",animations
speicher&=animations*5100
s=FRE(0):PRINT "Frei";s:PRINT "Benötigt";speicher&
IF speicher&-2000<s THEN
 CLEAR ,speicher&
 PRINT "Speicher vergroessert":PRINT "Free";FRE(0):END
ELSE
 PRINT "Speicher reicht"
END IF
INPUT "Warteschleifenlaenge  ",zeit
animations=animations-1:CLS
DIM pic%(2500,animations)

boxer:
LINE (10,10)-(120,120),,b

painter:
IF INKEY$=" " THEN i=i+1:GOTO chaos
IF MOUSE(0)<>:0 THEN PSET(MOUSE(1),MOUSE(2))
GOTO painter

chaos:
GET (10,10)-(120,120),pic%(0,i)
IF i=animations THEN GOTO main
CLS:GOTO boxer

main:
FOR i=0 TO animations
PUT (10,10),pic%(0,i),PSET
FOR j=0 TO zeit:NEXT
NEXT
GOTO main
```

Sie sehen bereits an den vorherigen Ausführungen und am Listing, wie einfach das funktioniert. Starten Sie das Programm nun. Zuerst fragt das Programm Sie nach der Anzahl der Bilder und berechnet daraus den nötigen Speicherplatz (zum Erforschen einer genauen Berechnungsroutine war ich zu faul, die vorliegende Routine entspringt der Methode »Pi mal Daumen«). Ist der vorhandene Speicherplatz dafür zu klein, wird erst einmal der Speicher mit dem CLEAR-Befehl angepaßt. Daraufhin wird das Programm allerdings abgebrochen, da wir mit CLEAR auch gleichzeitig alle Variablen

und damit die Anzahl der Animationsphasen gelöscht haben. Sie müssen also das Programm zweimal starten, wenn der Platz nicht gereicht hat. Beim zweiten Start ist der Speicher bereits angepaßt, die Meldung »Speicher reicht« wird ausgegeben und die Größe der Warteschleife abgefragt. Ich empfehle Werte zwischen 100 und 2000, je nach Anzahl der Bilder und gewünschter Geschwindigkeit. Nun können Sie nach Herzenslust im ersten Bild mit der Maus herummalen. Der mit dem Rechteck markierte Bereich wird mit GET aus der Grafik genommen, wenn die Space-Taste gedrückt wird. Jetzt folgt das zweite Bild, und so weiter. Ist das letzte Bild gemalt, erfolgt nach dem Druck auf die Space-Taste die Animation.

Wählen Sie einmal 3 Animationsschritte an. Als Geschwindigkeit nehmen Sie 1000. Zeichnen Sie in das Bild ein Gesicht, das sich nach links wendet und den Mund nach unten verzieht. In das zweite Bild zeichnen Sie das gleiche Gesicht, aber mit geradem Mund und nach vorn gewendet. Im dritten Bild nun ein Gesicht, das nach rechts sieht und lächelt. Und – voila – schon haben wir unsere erste Animation perfekt: Jemand der seinen Blick nach rechts wendet und erfreut auf etwas blickt.

Anstatt mit einer primitiven Zeichenroutine wie unserem schon öfters verwendeten »painter« können wir die einzelnen Grafiken aber auch mit Befehlen wie LINE und CIRCLE aufbauen. Oder wir malen mit einem professionellen Werkzeug wie DeluxePaint, benutzen das IFF-Ladeprogramm aus dem Beispiele-Teil, und holen uns dann den gewünschten Teil der Grafik mit GET, um ihn später wiederzuverwenden. So könnten wir beispielsweise einen Adler malen, der mit seinen Schwingen schlägt. Verknüpfen wir die Art der Animation, wie wir sie hier beschrieben haben, mit dem vorher entwickelten Bewegungsprogramm für Arrays (Routine »Arraymove«), so können wir unseren Adler über das Bild flattern lassen.

Wie schon erwähnt, liegen unsere Möglichkeiten vor allem in der Kombination der verschiedenen Techniken, die Sie hier nur einzeln kennenlernen. Eine weitere Kombinationsmöglichkeit würde uns das Programm »3D-Grafik« bieten, das ebenfalls im Beispiele-Teil zu finden ist. Wir erstellen damit ein Objekt, drehen es um die gewünschten Achsen, und holen uns jede gewünschte Phase mit einem GET-Befehl aus der Grafik. Diese Phasen verwenden wir wie im obigen Listing als Einzelbilder eines Films. Damit sind wir an unserem Traumziel, der schnellen 3D-Animation, bereits ange-

kommen. Und das einfacher, als Sie wahrscheinlich dachten! Sicher, unsere 3D-Grafik ist nicht in Realzeit entstanden – was in BASIC auch gar nicht geht, wie Sie am Punkt »Winkel stetig verändern« des 3D-Programmes sehen werden – aber unser Objekt dreht sich, und das sogar recht schnell.

Doch obwohl wir schon am Ziel unserer Wünsche angekommen sind, brauchen wir noch lange nicht aufzuhören. Die Befehle PUT und GET bieten noch mehr als man ahnt. Eine Möglichkeit des PUT-Befehls ist es zum Beispiel, ein Bild effektvoll einzublenden. Ich spreche hier nicht davon, das Bild hereinzuscrollen – das haben wir bereits kennengelernt. Wenn Sie aufgepaßt haben, wissen Sie ja bereits, daß die ersten zwei Werte in einem Integer-Array die x- und y-Größe darstellen. Das wollen wir doch ausnutzen, und so erstellen wir das folgende Programm.

Laden Sie dazu das »PICTURE«-Demo der BASICdemos, löschen Sie alles hinter dem GET-Befehl und hängen Sie stattdessen das folgende Listing hinter das Restprogramm.

```
variablenset:
x=p(0):y=p(1)
main:
WHILE ((a <= x) AND (b <= y))
 PUT(1,1),p,PSET
  a = a + 2: b = b + 2
  p(0) = a: p(1) = b
WEND
```

Das Bild erscheint mit »magischem Flimmern«, wenn man es einmal dramatisch ausdrückt. Sie dachten sicher, es passiert etwas ganz anderes. Aber es erscheint nicht etwa ein Teil des Bildes, wenn die x- und y-Werte kleiner sind als beim Einlesen, sondern stattdessen nur ein wüstes Etwas. Woran liegt das? Ganz einfach, der PUT-Befehl holt nicht die x-Variablen in x-Richtung und y-Variablen in y-Richtung, sondern einfach die Anzahl hintereinanderliegender Variablen, die sich aus der Formel x*y*Bitplanezahl ergibt. Um das Bild wirklich schön stückweise erscheinen zu lassen, müßten wir also das gesamte Variablenfeld umstellen, was so viel Zeit kostet, daß wir es lieber so belassen, wie es jetzt ist.

Damit wären wir am Ende des GET- und PUT-Spektakels. Zum Schluß des Kapitels als Erfolgserlebnis noch eine kleine Änderung

am Programm, die den Triumph des Programmierers über die Technik wieder zum Vorschein bringt: Schreiben Sie zusätzlich in die erste Zeile nach *variablenset* noch folgende Anweisung

```
p(2)=0.
```

Ein effektvollerer Absturz ist Ihnen gewiß noch nicht begegnet! (erprobt auf amerikanischem Amiga unter Kickstart 1.1. Was bei Kickstart 1.2 passiert, weiß ich nicht.) Sie müssen nun den Amiga neu starten.

13.2.3 Schnelle Polygone

Für Realzeitanimation brauchen wir Geschwindigkeit. Einer der wenigen wirklich schnellen BASIC-Befehle ist der AREAFILL-Befehl, oder besser gesagt das Polygonzeichnen mit Hilfe von AREAFILL. Das Polygon muß dabei zunächst mit den AREA-Befehlen festgelegt werden, um anschließend mit AREAFILL auf dem Bildschirm zu erscheinen.

Wie das funktioniert, wissen Sie ja bereits. Nun wollen wir die Polygone zudem aber auch noch für unsere Animation einsetzen. Die Polygonbefehle bieten uns dazu allerdings nicht so viele Möglichkeiten wie der PUT-Befehl. Hier geht es also nicht darum, möglichst viele Möglichkeiten auszuschöpfen, sondern die wenigen gegebenen Möglichkeiten optimal zu nutzen. In diesem Fall bleibt uns nichts anderes übrig, als uns entweder auf unseren schöpferischen Geist zu verlassen, oder die Mathematik wieder ins Spiel zu bringen.

Versuchen wir es erst einmal mit der künstlerischen Methode: Probieren, bis das Ergebnis zufriedenstellend ist. Bei mir kam dabei das folgende Programm heraus.

```
cuberead:
 FOR i=0 TO 7:READ a(i):NEXT
 FOR i=0 TO 7:READ x$(i):NEXT
 AREAFILL:CLS
```

```
cube:
 CLS
 FOR i=0 TO 7 STEP 2
 IF a(i)<0 OR a(i+1)<0 THEN END
 IF a(i)>WINDOW(2) OR a(i+1)>WINDOW(3) THEN END
 AREA (a(i),a(i+1))
 NEXT
 AREAFILL
warte:
 IF MOUSE(0)=0 THEN GOTO warte
  FOR i=0 TO 7
   IF x$(i)="+" THEN
    a(i)=a(i)+2
   ELSE
    a(i)=a(i)-2
   END IF
  NEXT
 GOTO cube
DATA160,60,250,60,250,100,160,100
DATA+,-,+,+,-,+,-,-
```

In diesem kleinen Beispiel habe ich einfach die Koordinaten der Anfangspunkte in DATA-Zeilen abgelegt. Die zweite DATA-Zeile gibt an, wie jeder Punkt geändert werden soll, wenn der Mausknopf gedrückt ist. Die IF-Befehle innerhalb der FOR-Schleife prüfen diese Bedingung und ändern die Position der Punkte (in diesem Beispiel um jeweils zwei Punkte). Ändern wir den Wert bei »+2« oder »–2«, dann erreichen wir dadurch eine Änderung sowohl der Bewegung als auch der Bewegungsgeschwindigkeit. Nehmen Sie nun einmal den CLS-Befehl nach dem Label »cube« aus dem Programm heraus und starten Sie neu. Das Ergebnis sieht nun schon ganz anders aus. Das Bild flimmert nicht mehr so, dafür aber bleiben die Reste des letzten Polygons zurück. Mit diesem »Nachzieher« können wir aber viele interessante Effekte erzielen. Bauen Sie zum Beispiel einen PATTERN-Befehl ein. Oder mehrere PATTERN-Befehle, die sich ihre Daten wieder aus DATA-Zeilen herausholen. Wenn Sie nun durch Anpassung der »+« und »-« das Bild noch so ändern, daß es sich langsam von der Mitte heraus an die Ecken des Windows hindreht, haben Sie schon einen schönen Einleitungsvorspann für ein Spiel (vorausgesetzt, Sie nehmen die Mausabfrage heraus). Das Ganze

noch garniert mit Farbe (ein COLOR-Befehl innerhalb der Schleife), und schon ist eine schöne farbenfrohe Animation entstanden.

Nun ist das Herumprobieren manchmal sehr mühselig; schließlich wollen wir ganz gezielt eine bewegte Grafik aufbauen. Stellen sie sich einmal vor, das Quadrat soll sich im Kreis drehen. Wie würden Sie das anfangen? Mit den Möglichkeiten des vorherigen Programms geht das nicht. Unsere DATA-Zeilen mit »+« und »–« geben im Zusammenwirken mit den Variablenzuweisungen nach den IF-Befehlen immer eine gerade Linie vor. Und hier kommt wieder die Mathematik ins Spiel. Diesmal verwenden Sie ganz einfach den Kreisalgorithmus aus dem ersten Buchteil. Ausnahmsweise zeige ich Ihnen hier nicht, wie es geht. Probieren Sie es selbst einmal aus!

Ein anderer Effekt, den wir aus unserem Kreis-Algorithmus ziehen können, ist der »Scheinwerfer«. Das folgende Programm benutzt eine vereinfachte Version des Kreisalgorithmus und einige wenige Berechnungen für die AREA-Befehle.

```
x0=320:y0=100:x=320:y=170
da=.01:a=3.14*2
GOSUB kurve
END

kurve:
angledrawn=0
xarc=x
yarc=y
 WHILE angledrawn < a
  angledrawn=angledrawn+da
  xarc=xarc+(y0-yarc)*da
  yarc=yarc+(xarc-x0)*da
   GOSUB poly
 WEND
RETURN

poly:
CLS
AREA (xarc,yarc)
AREA (x,y)
AREA (x0,y0)
AREAFILL
RETURN
```

Sie werden bemerken, daß das Bild stark flimmert. Das liegt daran, daß wir nach jeder Phase der Animation das Bild löschen. Statt mit CLS könnten wir das bereits gemalte Polygon aber auch einfach mit einem schwarzen Polygon übermalen. Die Geschwindigkeit sinkt dadurch aber wieder etwas, und das Flimmern ist trotzdem wieder da. Eine Methode, das Flimmern zu entfernen, nennt man »Double Buffering«. Leider funktioniert sie in AmigaBASIC nicht so recht.

Double-Buffering bedeutet, daß wir immer zwischen zwei Bildern hin- und herschalten. So sieht der Betrachter nie den Aufbau oder das Löschen des Bildes, sondern immer nur das jeweils fertige Bild. Und so wird es gemacht: Zwei Screens werden geöffnet. Die Grafik wird zunächst im nicht sichtbaren Screen gelöscht und neu aufgebaut. Erst wenn das Bild fertig ist, wird dieser Screen sichtbar gemacht (nach vorne geholt). Während man das flimmerfreie Bild ansehen kann, wird bereits im zweiten (dahinter liegenden) Screen die nächste Phase aufgebaut, dann gezeigt und so weiter.

AmigaBASIC erlaubt uns das nicht, denn erstens wird jedesmal, wenn wir auf einen anderen Screen umschalten, der Screen gelöscht, und zweitens würde dieser Vorgang in BASIC so viel Zeit benötigen, daß der Effekt des Double Buffering – flimmerfreie Grafik – wieder zunichte gemacht wird. Das Betriebssystem des Amiga verfügt aber über einen sogenannten »Double-Buffering«-Modus. Wird er angeschaltet, braucht sich der Programmierer nicht mehr um das Umschalten der Screens zu kümmern. Das macht das Betriebssystem dann ganz von selbst. Nur leider nimmt BASIC davon keine Notiz, außer wenn man Library-Funktionen verwendet. Und dazu müßten wir schon jetzt in die Tiefen der Amiga-Betriebssystem-Programmierung tauchen.

Mit einigem Geschick in der Mathematik der 3D-Algorithmen (siehe Buchteil 1) können Sie mit Hilfe der Polygonbefehle einen dreidimensionalen Würfel erstellen. Bei Einsatz des COLOR-Befehls an den richtigen Stellen haben Sie dann sogar einen 3D-Würfel mit Schattierungen. Es ist alles nur eine Sache der Mathematik und nicht etwa des Programmierens.

Wie man mit Polygonen in einer professionelleren Programmiersprache als BASIC 3D-Grafik mit ausgefüllten Flächen erzeugt, zeigt das Public-Domain-Demo »Amiga 3D« von Bart Whitebook. Bart Whitebook ist ein Mitarbeiter von Commodore-Amiga, der

bereits durch Computeranimation im Zeichentrickfilm »Das Geheimnis von Nimh« Erfahrungen sammelte. Jeder Amiga-Besitzer, der sich für schnelle Grafik interessiert, sollte sich dieses Programm einmal ansehen (Fish-Disk 17)

13.2.4 Farbanimation

Echte Grafikfans haben sich sicher bereits DeluxePaint oder Deluxe Paint II gekauft. Wenn man in diesem Programm die TAB-Taste drückt, beginnt die sogenannte »Farbanimation«. So etwas Ähnliches wollen wir nun auch programmieren. Für die Bestimmung der Farben haben wir im AmigaBASIC den PALETTE-Befehl. Und den wollen wir jetzt fleißig nutzen. Schließlich kann man damit ja nicht nur Farben durchlaufen lassen. Man kann auch Bilder ein- und ausblenden, indem man durch Farbänderungen ihre Helligkeit langsam größer oder kleiner werden läßt. Da das Ein- und Ausblenden das leichtere Unterfangen ist und wir immer den Weg des geringsten Widerstandes gehen, widmen wir uns zuerst diesem Problem. Es wird durch die folgende Routine gelöst.

```
Einblenden:
FOR i%=0 TO 2^Planes-1
 PALETTE i%,0,0,0
NEXT
PUT (0,0),pic%,PSET
FOR Br=14 TO 1 STEP -1
 FOR i%=0 TO 2^Planes-1
 PALETTE i%,col%(i%(*3)/Br,col%(i%*3+1)/Br,col%(i%*3+2)/Br
 NEXT
 FOR i%=0 TO pause:NEXT
NEXT
```

Alleinestehend wird das so natürlich nicht funktionieren. Sie müssen diese Routine in ein größeres Rahmenprogramm einbauen, in dem mit der entsprechenden Laderoutine (diese ist im Beispiele-Teil zu finden) das Bild zunächst von der Diskette gelesen wird. Das Bild liegt danach im Integer-Feld »pic%«. Die Variable »Planes« gibt an, wieviele Bitplanes die Grafik tief ist. Dazu passend müssen wir natürlich dann mit dem SCREEN-Befehl einen entsprechenden SCREEN öffnen. Die Farben liegen im Variablenfeld col%, und zwar mit jeweils drei Farbanteilen für eine Farbe: col%(0) ist der Rot-

anteil von Farbe 1, col%(1) der Grünanteil, col%(2) der Blauanteil von Farbe 1. Der Rotanteil von Farbe 2 liegt dann in col%(3), und so weiter. Der Wert »pause« gibt die Länge der Warteschleife an, die zwischen jedem Farbwechsel eingelegt wird.

Die Funktionsweise des Einblendens ist denkbar einfach: Zuerst setzen wir alle Farben auf Schwarz. Dann steigern wir gleichmäßig die Rot-, Grün- und Blau-Anteile einer Farbe und das Bild erscheint. Das gleiche können wir nun auch umgekehrt machen. Diesmal geht die Schleife »Br« (»Br« für »Bright«, also Helligkeit) nur andersherum, also von 1 bis 15. Dabei brauchen wir auch nicht vorher alle Farben auf Schwarz setzen. Wir lassen sie so, wie sie sind und »schrauben« sie einfach langsam ins Schwarz hinunter. Ist das ganze Bild ausgeblendet, also schwarz, können wir das nicht mehr benötigte Bild löschen; zunächst mit *CLS* den Bildschirm und dann mit *ERASE pic%* die entsprechende Feldvariable. Und schließlich sollten wir auch die Farben wieder zurücksetzen.

Doch nun zur Farbanimation. Hierbei kommt es in erster Linie darauf an, die Bilder so zu malen, daß aus dem Farbwechsel auch die Illusion von Bewegung wird. Das können Sie wieder mit einem professionellen Programm wie DeluxePaint oder GraphiCraft machen. Oder sie verwenden zum reinen Herumprobieren eine kurze Routine wie den oben beschriebenen »painter«. Alles, was wir jetzt noch machen müssen, ist der Wechsel der Farben. Dazu gibt es allerdings auch mehrere Möglichkeiten. Die erste ist die, die auch »GraphiCraft«, »DeluxePaint« und »Aegis Images« benutzen: Farbe 1 wird auf Farbe 2 kopiert, Farbe 2 auf Farbe 3,.....Farbe 31 auf 32, 32 wieder auf 0. So entsteht der Eindruck der Änderung vorhandener Farben. Die Bereiche, in denen herumkopiert wird, kann man natürlich auch entsprechend ändern, und so mehrere Farbanimationseffekte gleichzeitig verwenden. Methode 2 ist die, die zum Beispiel der »Aegis Animator« verwendet: Eine einzelne Farbe ändert ständig ihren Wert, alle anderen Farben bleiben gleich. So kann man jede einzelne Farbe animieren und damit noch eindrucksvollere Effekte erzielen. Dafür aber wird es schwieriger, Bilder so zu malen, daß der Eindruck von Bewegung entsteht. Sie haben also die Wahl zwischen der Einfachheit der Bilderstellung und der Opulenz der optischen Eindrücke. Ich persönlich bevorzuge die erste Methode. Und deswegen werden wir diese nun auch miteinander durchgehen.

Zur Farbrotation müssen alle Farben um jeweils eine Position verschoben werden. Dazu speichern wir die aktuelle Farbbelegung in einem Variablenfeld ab und kopieren dieses Variablenfeld dann – um eine Indexposition versetzt – in ein anderes Feld. Dann ändern wir die Farben mit der PALETTE-Anweisung entsprechend den Inhalten des zweiten Variablenfeldes, können dann wieder den aktuellen Farbinhalt im ersten Feld zwischenspeichern, und das Spielchen beginnt von Neuem.

In unserem Beispiel wollen wir der Einfachheit halber nur Graustufen verwenden. Da bei Grau die Anteile von Rot, Grün und Blau immer jeweils gleich hoch sind, benötigen wir nur eine Variable für alle drei Farbanteile. Zuerst müssen wir natürlich einen SCREEN öffnen, der den Farberfordernissen unserer Anwendung entspricht. In unserem Beispiel ist dies ein 5-Bitplane-Screen im LoRes-Modus (320x200 Punkte).

```
DIM col(31),x(31) 'x=Zwischenspeicher,col=anzuzeigende Farbe
SCREEN 1,320,200,5,1
WINDOW 2,,,,1
FOR i=0 TO 31
 col(i)=i/31  'Helligkeitsanteil jeder Farbe bestimmen
PALETTE i,col(i),col(i),col(i)
NEXT
z=-1          'mit Zeile 0 anfangen (z=-1, weil nach z=z+1 z=0)
FOR i=0 TO 200
z=z+1
IF z=31 THEN z=0        'Farbe für LINE
LINE (0,i)-(320,i),z    'Grafik zeichnen
NEXT

farbscroll:
 FOR z=0 TO 100         '100mal scrollen
  FOR v=0 TO 31
   x(v)=col(v)          'Zwischenspeicher=alte Farbe
  NEXT
   col(0)=x(31)         'neue Farbe 0=alte Farbe 31
     FOR v=1 TO 31
      col(v)=x(v-1)     'neue Farbe=alte vorherige Farbe
     NEXT
```

```
  FOR i=0 TO 31
   PALETTE i,col(i),col(i),col(i)  'Farbe setzen
  NEXT
 NEXT
RETURN
```

Wenn Sie das Programm starten, sehen Sie, daß das Farbscrolling sehr langsam ist. Das liegt nun leider wieder an der Interpretersprache BASIC. (In C oder Maschinensprache ist es möglich, die Farben mit großer Geschwindigkeit zu ändern.) Zudem können wir in BASIC den Farbwechsel nicht »nebenher« laufen lassen. Dazu gibt es hier eine andere Möglichkeit: Wir binden ein Maschinenspracheprogramm mit dem »LIBRARY«-Befehl ein und starten es. Die Zeiger auf die entsprechende Intuition-SCREEN-Struktur, in der weitere Zeiger auf Farbtabellen liegen (die es zu ändern gilt), finden wir mit den entsprechenden Befehlen heraus (vergleichen Sie dazu im Beispielteil das Listing »Bildschirmspeicher und SCREEN-Zeiger«).

Nun zu Methode 2, die wir gleich mit einem neuen Aspekt verknüpfen werden. Wir ändern nur eine einzige Farbe. Der neue Aspekt dabei ist, daß wir diesmal die Farbänderung in DATA-Zeilen ablegen. So müssen wir nicht jedesmal die alte Farbe zwischenspeichern. Der Vorgang wird dadurch viel schneller. Das ist so einfach, daß Sie es selbst als BASIC-Anfänger ohne Probleme zustandebringen, deswegen will ich hier keinen Platz für ein Listing verschwenden.

Dieselbe Methode ohne Data-Zeilen ist ebenfalls einfach. Wir nehmen Farbe 1 und speichern die anderen Farben der Reihe nach in Farbe 1. Oder wir berechnen den Farbwechsel für die zu ändernde Farbe unabhängig von den anderen Farben, wenn wir nur eine Farbe verdunkeln oder erhellen wollen. Das zeigt das folgende Beispiel.

```
COLOR 2
AREA (10,10):AREA (200,100):AREA (80,170)
AREAFILL
v=1
```

```
WHILE 1
 u=0:w=31:g=g+1
 IF g/2=INT(g/2) THEN v=-v:SWAP u,w
  FOR i=u TO w STEP v
   PALETTE 2,i/31,i/64,i/64
  FOR z=0 TO 50:NEXT
          NEXT
WEND
```

Hier ändern wir nur eine einzige Farbe, vom dunklen Schwarz zum schönsten Rosa. Die Teilwerte im PALETTE-Befehl können nach eigenem Gutdünken beliebig geändert werden, so daß beispielsweise ein gräßliches Grün zum leuchtenden Orange wird und so weiter. Diesmal haben wir keinen eigenen Screen benötigt, denn wir brauchen nicht so viele Farben, zwischen denen wir wechseln. Statt dessen ändern wir nur die Farbanteile einer Farbe. Das spart Speicherplatz und kann manchmal zu ebenso interessanten Effekten führen wie die andere Art des Farbscrollings.

Damit wären wir nun – fast – am Ende der Animationstechniken angelangt. Doch zuerst sehen Sie sich bitte den Abschnitt 13.3 an. Hier finden Sie einige größere Programmbeispiele. Auf diese Beispiele werden wir im darauffolgenden Kapitel zurückgreifen. Kapitel 13.4 beschäftigt sich mit der Zusammenfassung aller bisher gelernten Techniken und der Nutzung der Beispielprogramme für eigene Animationsprogramme. Ich werde versuchen, darin so viele Animationselemente wie möglich gleichzeitig zu nutzen.

13.3 Beispiele und nützliche Tools

In diesem Abschnitt werden Sie einige BASIC-Beispielprogramme kennenlernen. Einige davon können Sie vielleicht in eigenen Programmen verwenden. Andere sind vielleicht als Hilfsmittel bei der Programmiereung ganz nützlich.

13.3.1 Der IFF-Lader

IFF bedeutet »Interchange File Format«, was man in etwa mit »Dateiformat für den Datenaustausch« übersetzen kann. Mit Hilfe dieses Standards ist es möglich, Bilder verschiedener Programme,

die sich an diesen Standard halten, auszutauschen. So kann man dann auch DeluxePaint-Bilder in Dokumente der grafikfähigen Textverarbeitungen einbauen (zum Beispiel Vizawrite) oder in Tabellenkalkulations-Programmen verwenden. Der IFF-Standard umfaßt außer Grafik aber auch noch Textfiles (mit verschiedenen Zeichensätzen und Stilangaben), Musikfiles (zum Beispiel arbeitet »Deluxe Music Construction Set« damit), Datenlisten und weitere Möglichkeiten. Wir befassen uns hier aber nur mit einem einzelnen Teil des IFF-Standards, nämlich der Möglichkeit, Bilder zu speichern und zu laden.

Bevor wir zum ersten Programm kommen, möchte ich das Format kurz erklären. Alle IFF-Files sind in sogenannte »Chunks« aufgeteilt. Ein Chunk ist ein Datenblock. In diesem Datenblock stehen zuerst Informationen über den Typ der darin enthaltenen Daten, dann die Länge des Chunks, und schließlich die Daten selbst. Am Anfang eines jeden IFF-Programms steht das Wort »FORM« (ausgedrückt durch vier Bytes in ASCII-Code (also hexadezimal 46 4F 52 4D). Haben wir beim Laden der ersten vier Bytes eines Files dieses Wort nicht gefunden, ist es kein IFF-File. Der FORM-Chunk umfaßt die gesamte Datei und enthält alle anderen Chunks als Bestandteile. Die nächsten vier Bytes nach »FORM« geben die Länge des Chunks, in diesem Fall also die Länge der gesamten Datei in Bytes, an. Für unser BASIC-Programm benötigen wir das nicht, denn wir können das Ende des Files auch mit der Funktion EOF feststellen. Danach kommt die Festlegung des Typs des nächsten Chunks. Im Falle eines Bildes folgt hier ein »ILBM«. ILBM bedeutet »Interleaved Bitmap«, was etwa mit »in Zeilen aufgeteilte Bitmap« übersetzt werden kann.

Die Aufteilung der Bitmap in einzelne Zeilen anstatt in hintereinandergespeicherte Bitplanes erfolgt aus einem ganz einfachen Grund. So können wir nämlich Objekte beliebiger Größe speichern und sie auch in Grafiken anderer Größe ohne Schwierigkeiten einbauen. Also können mit dem ILBM-Format auch Sprites, BOBs und Teilbilder abgespeichert werden. Unser BASIC-Listing verwendet allerdings nur ganze Bilder. Nach den Bytes »ILBM« folgt als nächster Block gleich der Text »BMHD« für »Bitmap Header«. Danach folgt wieder die Länge des Chunks. Der BMHD-Chunk enthält die folgenden Daten: Breite und Höhe des Bildes in Pixels (je 1 Wort), die x- und y-Startposition des Bildes (je ein Wort), die Anzahl der Bitplanes, ein Byte für »Masking« (ob die Hintergrundfarbe

im Bild als transparent aufgefaßt werden soll), ein Byte für Datenkompression, ein Nullbyte, ein Wort, in dem die Farbe steht, die als transparent aufgefaßt werden soll, zwei Bytes, die das Verhältnis Breite zu Höhe der Pixel des Bildes angeben und die letzten beiden Bytes geben die gewünschte Screen-Breite und -Höhe an. Davon brauchen wir für unser Programm nur einige wenige Informationen (siehe Listing). Anschließend folgt der CMAP-Chunk. CMAP steht für »Colormap« und beinhaltet die RGB-Anteile der einzelnen im Bild verwendeten Farben. Anschließend kommt der CRNG-Chunk. »CRNG« steht für Color-Range und beinhaltet Daten für die Farbanimation. Und der BODY-Chunk enthält schließlich alle Grafikdaten. Genauer eingehen auf den gesamten Standard möchte ich nicht, denn er ist einfach zu umfangreich. Begnügen Sie sich deshalb bitte mit dem folgenden Listing.

Das Programm setzt DeluxePaint-Bilder (oder auch Bilder aus GraphiCraft oder Aegis Images) in ein Datenformat um, das Sie in einer einfachen FOR-NEXT-Schleife in ein Variablenfeld einlesen können. Das Variablenfeld brauchen Sie dann nur noch mit dem PUT-Befehl auf den Bildschirm setzen – und schon befindet sich dort eine wunderschöne Grafik.

Beim Ausdruck auf meinem Drucker hatte das folgende Listing nur zwei Seiten, es ist also keine unmenschliche Arbeit, alles abzutippen. Sie müssen nur aufpassen, daß Sie kein Komma, kein Prozentzeichen und keinen Punkt vergessen. Daraus hat sich beim Programmieren schon eine Menge Fehler ergeben. Ich arbeite mit so vielen Short-Integer-Variablen (%), um Speicherplatz zu sparen. Eine LoRes-Grafik (320 x 200) benötigt immerhin 40 KByte Speicher, HiRes-Grafiken entsprechend mehr. Aus Speicherplatzgründen können leider nicht alle IFF-Bilder eingelesen werden. 320 x 200 Pixels große Bilder können noch in allen Farben dargestellt werden. Im Interlace-Modus (320 x 400) sind 5 Bitplanes nur noch dann möglich, wenn der CLEAR-Befehl am Anfang des Programms auf 160 000 hochgesetzt wird. Die Verwendung einer solchen Grafik für eigene Programme ist dann kaum noch möglich. Grafiken in 640 x 200 dürfen bis zu 4 Bitplanes haben (also 16 Farben), sehr hochauflösende Bilder (640 x 400) nur noch 2 Bitplanes (4 Farben).

```
REM IFF-Bilder
REM Lassen Sie die Kommentare beim Eintippen weg (Speicher!)
IF FRE(0) < 140000& THEN CLEAR ,150000&  'Speicher freigeben
ON ERROR GOTO Fehler2
INPUT " Bitte Name eingeben:";Picture$
OPEN Picture$ FOR INPUT AS 1
IF INPUT$(4,1)   "FORM" THEN Fehler  'Kein FORM - kein IFF
x$=INPUT$(4,1)  'Chunklaenge interessiert nicht
IF INPUT$(8,1)<>"ILBMBMHD" THEN Fehler  'Kein Bild x$=INPUT$(4,1)  'Chunk
ist uns egal
Breite%=CVI(INPUT$(2,1))
Hoehe%=CVI(INPUT$(2,1))
x$=INPUT$(4,1) 'Startposition ist egal, denn wir laden Bild,
                'nicht Sprite
Planes%=ASC(INPUT$(1,1)) 'Anzahl der Bitplanes
x$=INPUT$(1,1)   'Nullbyte übergehen
Compression%=ASC(INPUT$(1,1)) 'Komprimiert, ja/nein?
x$=INPUT$(9,1) 'Rest des Chunks interessiert uns nicht
IF INPUT$(4,1)<>"CMAP" THEN Fehler
x$=INPUT$(4,1) 'Chunklaenge
Modus%=Breite%/320+(Hoehe%/200-1)*2          'Modus für Screen-Befehl
x$="Muss Juergen wuergen. Bitte warten..."   'Text für Window
SCREEN 1,Breite%,Hoehe%,Planes%.Modus%       'Screen an Bild anpassen
WINDOW 2,x$,,1,1                     'neues Output-Window auf Screen 1
DIM col%(2^Planes%-1)
FOR t%=0 TO UBOUND(col%)
 col%(t%*3)=ASC(INPUT$(1,1))/16
 col%(t%*3+1)=ASC(INPUT$(1,1))/16
 col%(t%*3+2)=ASC(INPUT$(1,1))/16
 PALETTE t%,col%(t%*3)/15,col%(t%*3+1)/15,col%(t%*3+2)/15
NEXT t%
Main:
 IF INPUT$(1,1)<>"B" THEN Main
 IF INPUT$(3,1)<>"ODY" THEN Main      'Suche nach Grafikdaten (BODY)
 Laenge&=CVL(INPUT$(4,1))                    'Laenge der Grafikdaten
DIM pic%((Breite%*Hoehe%)/16*Planes%+3)
pic%(0)=Breite%
pic%(1)=Hoehe%
pic%(2)=Planes%
t&=1:x$="":Zeile%=0:Seite%=0:Bytes%=0 'Zaehler auf Anfang,
```

```
              'Grafikdatenstring auf Null,Grundlagen zur
              'Variablenfeldberechnung für GET/PUT-Bild
Groesse%=Breite%*Hoehe%/8:Spalten%=Breite%/8        'Bitplanegroesse
                   'in Bytes, Breite in Bytes als Umrechnungsgrundlage
IF Compression%=0 THEN
 GOSUB Normal  'Wenn ungepackt, dann einfache Laderoutine
ELSE
 GOSUB Gepackt 'Wenn gepackt, dann laden und entpacken
END IF
CLOSE 1

Schluss:
WINDOW 3,,(10,20)-(120,60),0,1
WINDOW OUTOUT 3
PRINT "RETURN = SAVE"
PRINT "SPACE  = STOP"
GOSUB Antwort
IF Antwort$=CHR$(13) THEN GOSUB Speichern
PRINT:PRINT "Ende.":WINDOW 1:END

Normal:
WHILE t&<Laenge&
 x$=INPUT$(2,1):t&=t&+2
 GOSUB Auswertung
WEND
RETURN

Gepackt:
WHILE t&<Laenge&
 Code%=ASC(INPUT$(1,1)):t&=t&+1
  IF Code% 128 THEN    'Wenn gepackte Zeile
   FOR r%=0 TO Code%
    x$=x$+INPUT$(1,1):t&=t&+1
    IF LEN(x$)=2 THEN GOSUB Auswertung
   NEXT
```

```
  ELSE    'Wenn Zeile nicht gepackt
    Zeichen$=INPUT$(1,1):t&=t&+1
    FOR r%=0 TO 256-Code%
      x$=x$++Zeichen$
      IF LEN(x$)=2 THEN GOSUB Auswertung
    NEXT
WEND
RETURN

Auswertung:
Index%=(Groesse%*Seite%+Spalten%*Zeile%)/2+Bytes%+3
Bytes%=Bytes%+1
IF Bytes%=Spalten%/2 THEN Bytes%=0:Seite%=Seite%+1
IF Seite%=Planes% THEN Seite%=0:Zeile%=Zeile%+1
IF Seite%=0 AND Bytes%=0 THEN PUT (0,0),pic%,PSET
pic%(Index%)=CVI(x$):x$=""
RETURN

Fehler:
CLS:PRINT CHR$(7)
PRINT "Kein IFF oder schlechter Versuch eines IFF-Files"
PRINT CHR$(13)+"Neustart mit RETURN"

Warte:
x$=INKEY$:IF x$="" THEN Warte
IF x$=CHR$(13) THEN RUN
GOTO Fehler

Fehler2
IFF ERR=53 THEN ERROR ERR
PRINT CHR$(12)
PRINT "Datei";Picture$;"nicht gefunden."
PRINT "Weiter mit Return oder Space"
GOSUB Antwort
RUN

Antwort:
Antwort$=INKEY$
IF Antwort$="" THEN Antwort
IF Antwort$=CHR$(13) OR Antwort$=" " THEN RETURN
GOTO Antwort
```

```
Speichern:
PRINT CHR$8(13)+"Farbinfo-SAVE?"+CHR$(13)
PRINT "RETURN =mit"+CHR$(13)+"SPACE  = ohne"
GOSUB Antwort
PRINT "Filename?":INPUT " ",x$
OPEN x$ FOR OUTPUT AS 1
FOR t%=0 TO UBOUND(pic%)
PRINT#1,MKI$(pic%(t%));
NEXT
IF Antwort$=CHR$(13) THEN
FOR t%=0 TO 2^Planes%*3-1
PRINT#1,CHR$(col%(t%));
NEXT
END IF
CLOSE 1
RETURN
```

Die Bedienung des Programmes ist recht einfach. Starten Sie es zunächst mit RUN. Wenn Sie ein HiRes-Bild laden wollen, müssen Sie vor dem Starten unter Umständen (wegen Speicherplatzproblemen!) das List-Window schließen und das Output-Window kleiner machen. Nach dem Start wird dann zunächst nach dem Filenamen gefragt. (Bitte vergessen Sie nicht, daß Sie vor Dateinamen, die sich auf Dateien beziehen, die nicht im aktuellen Directory liegen, den Laufwerksnamen und eventuelle Subdirectories angeben müssen.) Das File wird jetzt geöffnet. Sie müssen dann eine ganze Weile warten, bis das Bild umgerechnet ist. Die jeweils berechnete Zeile wird immer dem Bild hinzugefügt, so daß Sie am Bildschirm verfolgen können, wie weit der Prozeß schon ist. Das Programm setzt das IFF-Format in eine Feldvariable um, die Sie sofort danach mit *PUT (x,y),pic%* auf den Bildschirm bringen können. Ist das Bild umgerechnet, erscheint ein weiteres Fenster. Drücken Sie RETURN, wenn Sie das Bild als Variablenfeld abspeichern wollen. Ein Druck auf die SPACE-Taste bricht das Programm ab. Der Screen mit der Grafik bleibt vorhanden, und auch die benutzten Variablen werden nicht gelöscht. Sie können damit nach Herzenslust herumspielen.

Nach dem Drücken von RETURN wird gefragt, ob Sie die Farben auch abspeichern wollen. Im Normalfall enthält ein Array, das man mit PUT auf den Bildschirm setzen kann, keine Farbinformationen. Haben Sie RETURN gedrückt, wird beim Speichern hinter das abge-

speicherte Array »pic%« noch die Farbinformation »col%« gespeichert. Nach dem zweiten RETURN wird noch der Filename abgefragt, bei dem Sie bitte wieder den Laufwerksnamen (z.B. »df1:«) und die entsprechenden Subdirectories nicht vergessen. Nach dem Speichern wird auf den Workbench-Bildschirm zurückgeschaltet. Wenn Sie dort mit der Maus in das Output-Window klicken, können Sie ganz normal weiterarbeiten. Alle Variablen sind noch vorhanden. Mit den Tastenkombinationen Amiga-N und Amiga-M (linke Amiga-Taste) können Sie zwischen den Screens umschalten, ohne irgendwelche Fenster verändern zu müssen.

Die wichtigsten Variablen des Programmes sind:

Picture$	Dateiname
x$	Variable zum Einlesen von Strings für die Umsetzung auf Array-Format, wird aber auch als Dummy für Windowname und Keyboardeingabe benutzt.
Breite%	Breite des Bildes in Pixels
Hoehe%	Höhe des Bildes in Pixels
Planes%	Anzahl der Bitplanes
Compression%	gleich Null, wenn ungepacktes Bild, sonst größer.
Modus%	Modus für den Screen-Befehl, wird aus Höhe und Breite errechnet.
col%()	Array von 0 bis 95 in folgender Reihenfolge:
col%(0)	Rot-Anteil von Farbe 0
col%(1)	Grün-Anteil von Farbe 0
col%(2)	Blau-Anteil von Farbe 0
col%(3)	Rot-Anteil von Farbe 1 ... und so weiter
t%,r%	Schleifenzähler für Berechnungen in FOR-Schleifen
Laenge&	Länge des BODY-Chunks, also Anzahl der Bytes, die Grafikdaten enthalten
pic%()	Variablenfeld, in das das Bild gespeichert wird, enthält erst drei Bytes für Breite, Höhe und Anzahl der Bitplanes, dann folgen die Grafikdaten.

t&	Zähler innerhalb der Umrechnung, wird mit "Länge&" verglichen.
Code%	Erstes Datenbyte einer Zeile. Wenn "Code%" kleiner als 128 ist, ist die Grafikzeile gepackt.
Zeichen$	Variable zum Lesen einzelner Bytes bei gepackten Zeilen.
Index%	Index des zur Zeit bearbeiteten pic%-Elementes
Bytes%	Anzahl der umgerechneten Bytes
Spalten%	Anzahl der Bytes pro Grafikzeile
Seite%	Bitplane, für die gerade ein Byte umgerechnet wird.
Antwort$	String für Tastaturabfragen

Mit Hilfe dieser Ausführungen und der im Listing abgedruckten Kommentare werden Sie sicherlich die Arbeitsweise des Programmes verstehen. Ich habe versucht, das Programm so universell wie möglich zu halten. Wenn Sie nur ungepackte Bilder für BASIC umrechnen lassen wollen (also mit Graphicraft abgespeicherte Bilder), können Sie die Routine »Gepackt« weglassen und im Hauptprogramm die entsprechende Abfrage löschen. Sollen nur LoRes-Bilder geladen werden, brauchen Sie die Höhe, Breite und so weiter nicht mehr berechnen. Sie können das Programm also auf Ihre eigenen Erfordernisse zurechtschneidern. Das hier gezeigte Listing enthält die Maximal-Lösung. So können Sie HiRes-Bilder auch mit mehr als zwei Bitplanes verwenden, wenn Sie das Programm so weit kürzen, daß es nur noch diese Bilder verarbeiten kann.

Natürlich gäbe es noch eine andere Methode, Bilder zu speichern, als die hier verwendete. Sie können dann in BASIC-Programmen schneller mit der Grafik arbeiten, denn mit einem einzigen PUT-Befehl ist die Grafik binnen Sekundenbruchteilen auf dem Bildschirm. Die andere Methode, von der ich spreche, ist es, die Grafik mit dem GET-Befehl vom Bildschirm in eine Feldvariable zu holen und diese dann – ohne Umrechnungen – in einer Datei zu speichern. Das so gespeicherte Variablenfeld kann man nun mit einer kleinen Laderoutine laden und mit einem PUT-Befehl auf den Bildschirm bringen. Diese Methode ist eindeutig schneller, aber nicht mehr kompatibel zu anderen Grafikprogrammen.

Ich habe hierzu wieder eine universelle Routine geschrieben, die verschiedene Auflösungen akzeptiert und sowohl Bilder mit als auch

ohne abgespeicherte Farbinformation lädt. Wenn Sie genau wissen, was für ein Bild Sie laden, kann die Laderoutine entsprechend gekürzt werden. Wenn Sie zum Beispiel genau wissen, daß Sie ein LoRes-Bild mit 5 Bitplanes laden wollen, an dessen Ende Farbinformationen gespeichert sind, sähe der Lader dafür wie folgt aus:

```
DIMpic%(20004),col%(95)
OPEN Name$ FOR INPUT AS 1 LEN=<Filebuffer>
pic%(0)=CVI(INPUT$(2,1))
pic%(1)=CVI(INPUT$(2,1))
pic%(2)=CVI(INPUT$(2,1))
FOR t%=3 TO 20003:pic%(t%)=INPUT$(2,1)):NEXT
FOR t%=0 TO 31
 col%(t%*3)=ASC(INPUT$(1,1))
 col%(t%*3+1)=ASC(INPUT$(1,1))
 col%(t%*3+2)=ASC(INPUT$(1,1))
NEXT
CLOSE 1
```

Nun sind alle nötigen Variablen geladen. Das Bild kann mit dem folgenden Programmteil dargestellt werden.

```
SCREEN 1,320,200,5,1
WINDOW 2,,,0,1
FOR t%=0 TO 31
 PALETTE t%,col%(t%*3)/15,col%(t%*3+1)/15,col%(t%*3+2)/15
NEXT
PUT (0,0),pic%
```

Sie sehen, daß wir sowohl im IFF-Umrechnungsprogramm als auch im Ladeprogramm als Windowstatus 0 verwendet haben. Das tun wir aus dem einfachen Grund, den sogenannten »Heap-Space« nicht zu überfordern. Smart-Refresh-Windows oder auch Gadgets benötigen alle viel Platz im »System-Heap«, einer Art Stapelspeicher des Amiga-Betriebssystems. Ist der Heap voll, »geht nichts mehr«. Und das wollen wir vermeiden. Am Anfang unseres kleinen Listings haben Sie sicher gesehen, daß wir nach dem OPEN-Befehl ein *LEN=<Filebuffer>* angefügt haben. Damit bestimmen wir die Größe des Filebuffers. Sie müssen dazu *<Filebuffer>* durch eine Zahl ersetzen. Im Normalfall ist der Buffer für das Laden von Short-Integer-Daten 512 Bytes groß. Erhöhen wir den Wert, verlieren wir zwar wertvollen Speicherplatz, dafür aber ist das Bild schneller im

Speicher. Durch die Erhöhung wird nämlich mehr von den Daten auf einmal gelesen und wir müssen nicht so oft auf die Diskette zugreifen. Sie müssen selbst erkennen, ob für Ihre eigene Anwendung der Speicher soweit ausreicht, daß noch Platz für eine Erhöhung des Filebuffers bleibt.

Beim Laden eines Bildes sollten Sie darauf achten, daß Sie vorher genügend Speicher mit dem CLEAR-Befehl für BASIC-Variablen (also unser Bild »pic%()« und die Farben »col%()«) reservieren. Ich schlage als guten Durchschnitt 150 000 Bytes vor. Das geschieht mit dem folgenden Befehl.

```
CLEAR ,150000&
```

Manchmal erfolgt daraufhin ein »Out of Memory«. Das kommt davon, daß eventuell vorher schon eine Menge mit BASIC gearbeitet wurde. BASIC räumt leider nicht allen »Müll« auf, den es im Speicher produziert hat, und auch der CLEAR- Befehl beseitigt leider nicht alles. Booten Sie in diesem Fall neu und starten Sie BASIC erneut. Wenn Sie auch die Workbench übergehen (wie oben beschrieben) und BASIC vom CLI aus laden, ist noch mehr Speicher frei. Mehr als 160 000 Bytes bekam ich aber in der Regel nie für BASIC.

Hier folgt aber nun unser universelles Ladeprogramm, das verschiedene Auflösungen und auch das Fehlen einer Farbinformation berücksichtigt.

```
INPUT "Filename ",x$
OPEN x$ FOR INPUT AS 1
x%=CVI(INPUT$(2,1)) 'Breite
y%=CVI(INPUT$(2,1)) 'Hoehe
z%=CVI(INPUT$(2,1)) 'Bitplanes
DIM col%(2^z%-1)    'Farbbedarf an Bitplanes anpassen
v=20004            'Anzahl der %-Variablen für geringste
                     'Auflösung
IF x%=640 THEN v=v*2 'Verdoppeln, wenn HiRes
IF y%=400 THEN v=v*2 'Verdoppeln, wenn Interlace
DIM pic%(v)          'Variable auf Bedarf dimensionieren
pic%(0)=x%:pic%(1)=y%:pic%(2)=z%       'PUT-Konventionen
FOR i%=3 TO v-1
 pic%(i%)=CVI(INPUT$(2,1)) 'Grafikdaten einlesen
NEXT
```

```
IF EOF(1) THEN              'Wenn File-Ende, kein Farbinfo
  INPUT "Ohne Farbinfo; Press RETURN",x$
ELSE
FOR i%=0 TO 2^z%-1                          'Farbdaten einlesen
 col%(i%*3)=ASC(INPUT$(1,1))
 col%(i%*3+1)=ASC(INPUT$(1,1))
 col%(i%*3+2)=ASC(INPUT$(1,1))
NEXT
END IF
```

Anzeige:

```
CLOSE 1
SCREEN 1,x%,y%,z%,(x%/320+(y%/200-1)*2)
WINDOW 2,,,0,1
FOR i%=0 TO 2^z%-1
 PALETTE i%,col%(i*3)/15,col%(i*3+1)/15,col%(i*3+2)/15
NEXT
PUT (0,0),pic%
END
```

Sie sehen, allzu lang oder kompliziert ist das Programm gar nicht geworden. Wenn Sie die Grafik nur als Hintergrund für ein eigenes Spiel mit Sprites verwenden wollen, empfiehlt sich, die Variablenfelder »pic%()« und »col%()« nach dem Einlesen und Anzeigen mit dem ERASE-Befehl zu löschen, um Speicherplatz zurückzugewinnen.

13.3.2 3D-Grafik

Doch nun weg von den gemalten Bildern und hin zu den mathematisch erzeugten. Sie haben bereits in den letzen Kapiteln und auch im Buchteil 1 gesehen, was man mit ein wenig Mathematik alles anfangen kann. Das folgende Listing stellt 3D-Grafiken dar, die Sie mit x-,y- und z-Koordinaten der Eckpunkte eingeben müssen. Das Programm kann Körper im Raum auf zwei Arten drehen: einmal um einen festen Winkel oder auch kontinuierlich unter Anzeige aller Zwischenstufen der Rotation.

Dieses Programm wurde von Dirk Dreyer geschrieben, bei dem ich mich recht herzlich dafür bedanke, daß er es mir zur Verfügung stellte. Wir Genies, die alle zufällig am selben Tag (und im selben

Jahr) geboren wurden, arbeiten doch immer wieder gerne zusammen. Nachfolgend nun das Listing des Programms. Dahinter finden Sie die Bedienung und Funktionsweise erläutert.

```
REM 3-d Grafik von Dirk Dreyer
REM px  = Koordinatenpunkt in x
REM py  = Koordinatenpunkt in y
REM  x  = wahre Laenge in x
REM  y  = wahre Laenge in y
REM  z  = wahre Laenge in z
REM a$  = Eingabevariable
REM b$  = Eingabeueberpruefung
REM a   = Laufvariable
REM asi = Sinus alpha
REM aco = Cosinus alpha
REM bsi = Sinus beta
REM bco = Cosinus beta
REM avo = von a
REM abi = bis zu
REM ml  = Vergroesserung
SCREEN 1,640,400,2,4
WINDOW 1,"Sience Screen",(0,0)-(631,300),8,1
WINDOW 2,"Input Screen  e-weiter n-neu",(0,300)-(631,380),8,1

Start:
WINDOW OUTPUT 1
CLS
PRINT
PRINT "                        3-D Grafik von Dirk Dreyer"
PRINT:PRINT:PRINT
PRINT" a..................3-D Grafik"
PRINT
PRINT" b..................zukuenftige Erweiterungen"
INPUT " Bitte waehlen Sie >";a$
IF a$ = "a" THEN Isometrie
IF a$ = "b" THEN Zukunft
WINDOW OUTPUT 2
CLS
GOTO Start
```

```
Isometrie:
WINDOW OUTPUT 1 : CLS : e=0 : a=0 : DIM py(300) : DIM px(300)
DIM x(300) : DIM y(300) : DIM z(300)

Iso:
WINDOW OUTPUT 1 : CLS
WINDOW OUTPUT 2 : CLS
PRINT "                         3 - D Grafik"
PRINT
INPUT " geben Sie Winkel alpha ein >";a$
IF a$ = "e" THEN Koerper
IF a$ = "n" THEN Iso
PRINT
GOSUB alpha
INPUT " Geben Sie Winkel beta ein >";a$
IF a$ = "n" THEN Iso
PRINT
GOSUB beta
CLS
PRINT                   " Format"
PRINT
INPUT" Geben Sie den Multiplikator an >";a$
IF a$ = "n" THEN Iso
ml = VAL (a$)

Schleife:
CLS
a = a+1 : e = e+1
PRINT " Punkt "a
PRINT
INPUT " x - Koordinate ";a$
IF a$ = "n" THEN a = a-1 : e = e-1 : GOTO Schleife
x(a) = VAL (a$) * ml
PRINT
INPUT " y - Koordinate ";a$
IF a$ = "n" THEN a = a-1 : e = e-1 : GOTO Schleife
y(a) = VAL (a$) * ml
PRINT
```

```
INPUT " z - Koordinate ";a$
IF a$ = "n" THEN a = a-1 : e = e-1 : GOTO Schleife
z(a) = VAL (a$) * ml
PRINT
GOSUB koordinaten
IF e < 2 THEN St
WINDOW OUTPUT 1
LINE (px(a),py(a))-(px(a-1),py(a-1)),3
WINDOW OUTPUT 2
e = 0
INPUT " Ende ";a$
IF a$ = "j" THEN Koerper
GOTO Schleife

St:
GOTO Schleife
Koerper:
WINDOW OUTPUT 2
CLS
PRINT" a.......................Winkel veraendern"
PRINT
PRINT" b.......................Zeichnung drehen"
PRINT
PRINT" c.......................Zeichnung laden"
PRINT
PRINT" d.......................Zeichnung speichern"
PRINT
INPUT " Bitte waehlen Sie >";a$
IF a$ = "a" THEN Winkel
IF a$ = "b" THEN Drehen
IF a$ = "c" THEN Laden
IF a$ = "d" THEN Speichern
IF a$ = "n" THEN Iso
IF a$ = "e" THEN Schleife
GOTO Koerper

alpha:
asi = SIN (3.141592*VAL(a$)/180)
aco = COS (3.141592*VAL(a$)/180)
RETURN
```

```
beta:
bsi = SIN (3.141592*VAL(a$)/180)
bco = COS (3.141592*VAL(a$)/180)
RETURN

koordinaten:
px(a) = 310-z(a)*aco+x(a)*bco
py(a) = 280 - z(a)*asi-x(a)*bsi-y(a)
RETURN

Speichern:
CLS
PRINT
INPUT " Geben Sie den Namen der Zeichnung an ";a$
IF a$ = "n" THEN Speichern
OPEN a$ FOR OUTPUT AS #1
WRITE #1,a,ml
FOR b = 1 TO a
 WRITE #1,px(b),py(b),x(b),y(b),z(b)
NEXT b
CLOSE #1
GOTO Koerper

Laden:
CLS
PRINT
INPUT " Name der Zeichnung >";a$
IF a$ = "n" THEN Laden
OPEN a$ FOR INPUT AS #1
INPUT#1,a,ml
FOR b = 1 TO a
 INPUT#1,px(b),py(b),x(b),y(b),z(b)
NEXT b
CLOSE #1
WINDOW OUTPUT 1
CLS
FOR b = 2 TO a STEP 2
LINE (px(b),py(b))-(px(b-1),py(b-1)),3
NEXT b
GOTO Koerper
```

```
Winkel:
CLS
PRINT
PRINT                    " Winkel veraedern "
PRINT
PRINT " a..............Winkel stetig veraendern"
PRINT
PRINT " b..............Winkel einfach veraendern"
PRINT
INPUT " Bitte waehlen Sie >";a$
IF a$ = "a" THEN Stetig
IF a$ = "b" THEN Einfach
IF a$ = "n" THEN Iso
GOTO  Winkel

Einfach:
CLS : INPUT "Winkel alpha?",a$ : GOSUB alpha
INPUT "Winkel beta?",a$ : GOSUB beta
WINDOW OUTPUT 1 : CLS
FOR b=1 TO a
 GOSUB koordinaten2:py(b)=py(b)+mx-y(b)
 LINE (px(b),py(b))-(px(b-1),py(b-1)),3
NEXT b
GOTO Koerper

Stetig:
CLS
PRINT
PRINT" a..................Winkel a und b gleichzeitig"
PRINT
PRINT" b..................Winkel a und b gegenseitig"
PRINT
INPUT " Bitte waehlen Sie >";b$
IF b$ = "a" THEN beidseitig
IF b$ = "b" THEN beidseitig
IF b$ = "n" THEN Winkel
GOTO Stetig
```

```
Beidseitig:
CLS
PRINT
INPUT " Der Winkel soll sich aendern von ";a$
PRINT
avon = VAL (a$)
INPUT " bis >";a$
abis = VAL (a$)
IF a$ = "n" THEN Stetig
WINDOW OUTPUT 1
CLS
FOR c = avon TO abis  STEP 3
  CLS
  LET a$ = STR$(c)
  GOSUB alpha
  IF b$ = "a" THEN GOSUB beta
  IF b$ = "b" THEN LET a$ = STR$ (abis-c) : GOSUB beta
  e = 1
  FOR b = 1 TO a
    IF c > (abis/2) THEN x = -1
    IF c < (abis/2) THEN x = 1
   ma = (c-abis/2)*x+abis/2
   mx = y(b)*3/abis*ma
   GOSUB koordinaten2
    IF e < 2 THEN En
   LINE (px(b),py(b))-(px(b-1),py(b-1)),3
    e=0

    En:
    e = e+1
 NEXT b
NEXT c
GOTO Koerper

Koordinaten2:
px(b) = 310-z(b)*aco+x(b)*bco
py(b) = 280-z(b)*asi-x(b)*bsi-mx
RETURN
```

```
Drehen:
CLS:INPUT "Wieviel Grad in x-Richtung?",a1
   INPUT "Wieviel Grad in y-Richtung?",a2
   INPUT "Wieviel Grad in z-Richtung?",a3
FOR b=1 TO a
 Drehungx:
 y(b)=y(b)*COS(a1)+z(b)*SIN(a1)
 z(b)=y*(-SIN(a1))+z(b)*COS(a1)
 Drehungy:
 x(b)=x(b)*COS(a2)+z(b)*(-SIN(a2))
 z(b)=x(b)*SIN(a2)+z(b)*COS(a2)
 Drehungz:
 x(b)=x(b)*COS(a3)+y(b)*SIN(a3)
 y(b)=x(b)*(-SIN(a3))+y(b)*COS(a3)
NEXT

IF VAL(a$) = 0 THEN a$="45"
GOSUB alpha : GOSUB beta
FOR b=1 TO a
  GOSUB koordinaten2
 py(b)=py(b)+mx-y(b)
NEXT
px(0)=px(1):py(0)=py(1)
WINDOW OUTPUT 1:CLS
FOR b=1 TO a
 LINE (px(b),py(b))-(px(b-1),py(b-1)),3
NEXT
GOTO Koerper

Zukunft:
PRINT "Noch nicht implementiert"
END
```

An diesem Listing kann man erkennen, daß man gute Grafikprogramme auch mit wenig Aufwand erstellen kann. Die ganze Grundlage des Programmes sind einige wenige Berechnungen, die in den Routinen »alpha«, »beta« und »Koordinaten« zu finden sind (Umrechnung der Koordinatenpunkte im Raum in Punkte in der Bildschirmebene »px« und »py« aus Betrachtungswinkel »alpha« und »beta«, mit Vergrößerung » ml«).

Nun zur Bedienung des Programms. Nachdem Sie es mit RUN gestartet haben, tippen Sie »a« (und RETURN) ein. Das »b« ist für zukünftige Erweiterungen gedacht. Der Autor arbeitet noch daran. Jetzt fragt das Programm nach Betrachtungswinkel »alpha«. Geben Sie daraufhin den Blickwinkel ein. Tippen Sie statt dessen »e« ein, wenn Sie gleich ins Hauptmenü wollen (zum Beispiel zum Laden von Bildern) oder »n« für nochmal. Anschließend geben Sie Betrachtungswinkel »beta« ein. Wenn Sie sich vertippt haben, können Sie auch hier mit »n« abbrechen und bei »alpha« wieder neu beginnen. Zu guter Letzt müssen Sie nun noch den Multiplikator eingeben. Je größer die Zahl, desto näher wird das Objekt herange»zoomt«.

Jetzt können Sie nacheinander die x-, y- und z-Koordinaten für jeden Eckpunkt des 3D-Objektes eingeben. Haben Sie sich vertippt, können Sie jederzeit mit »n« abbrechen, um mit der x-Koordinate nochmal zu beginnen. Nach jeder z-Koordinate müssen Sie bei der Frage »Ende?« mit einem anderen Buchstaben als »j« antworten, um einen weiteren Punkt anzufügen. »J« bringt Sie zurück ins Hauptmenü. Die Menüpunkte »Laden« und »Speichern« brauche ich wohl nicht mehr näher zu beschreiben. Wählen Sie »Winkel verändern« an, können Sie das Bild mit einem weiteren Menü entweder stetig oder in einem Schritt drehen. Bei »einfach« ändern Sie einfach nur den Betrachtungswinkel, um sich das Objekt von einem anderen Standort aus zu betrachten. Bei »stetig« müssen Sie einen Start- und einen End-Winkel angeben. Das Objekt dreht sich dann um die (zwischen den) angegebenen Winkel. BASIC ist allerdings so langsam, daß man dem Aufbau der Objekte ohne weiteres folgen kann. Könnten wir mit BASIC »Double Buffering« nutzen, wäre die Geschwindigkeit ausreichend, um eine einigermaßen fließende 3D-Grafik zu erzeugen. Doch das ist leider nicht möglich (siehe oben).

Beim Drehen eines Objektes können Sie wählen zwischen »gleichzeitig« und »gegenseitig«. Gleichzeitig bedeutet, daß der alpha- und der beta-Winkel gleichzeitig regelmäßig zunehmen. Bei »gegenseitig« ändern sich die Winkel in verschiedenen Drehrichtungen.

Sie können zum Beispiel 3D-Grafiken mit diesem Programm berechnen lassen, um sie dann in eigenen Programmen zu verwenden. Dazu gibt es verschiedene Möglichkeiten: Entweder Sie speichern nur die Realkoordinaten »px« und »py« der berechneten Punkte, um mit LINE-Befehlen im eigenen Programm die Grafik selbst zu erzeu-

gen, oder Sie holen sich das Aussehen der einzelnen Drehphasen mit dem GET-Befehl aus dem Bildschirm, um es bei Bedarf später mittels PUT auf den Bildschirm zurückzubringen.

13.4 Zusammenfassung

Sicher erwarten Sie jetzt, daß ich Ihnen erkläre, wie Sie gleichzeitig mehrere Sprites und BOBs über den Bildschirm bewegen, während ein mit DeluxePaint gemaltes Bild im Hintergrund vorbeiscrollt und dann auch noch gleichzeitig ein Spiel abläuft. Das funktioniert so leider nicht. BASIC ist eine viel zu langsame Sprache, um solche Effekte erzielen zu können. Es geht also nicht immer so einfach, wie man es sich vielleicht denkt. Ich will Ihnen nun aber zeigen, wie Sie das Beste herausholen können – das Beste aus BASIC und das Beste aus den bisher erlernten Techniken.

13.4.1 Wie scrolle ich ein ganzes Bild?

Zum Scrollen eines Bildes gibt es mehrere Möglichkeiten. Welche davon Sie verwenden, hängt immer ganz davon ab, was Sie machen wollen.

Ich nehme an, sie wollen eines der Bilder über den guten alten Röhrenbildschirm scrollen lassen, das Sie selbst (oder jemand anders) in mühevoller Kleinarbeit mit DeluxePaint oder GraphiCraft gemalt haben. Zuerst müssen Sie das Bild mit dem IFF-Wandler in ein Format umwandeln, das Sie vom BASIC aus verarbeiten können. Der IFF-Wandler speichert das Bild als sequentielle Datei ab, die – hintereinander gelesen – ein Variablenarray ergibt, das direkt mit PUT in den Bildschirm gesetzt werden kann. Und daraus ergibt sich schon Möglichkeit 1: Wir scrollen das Bild einfach mit dem PUT-Befehl über den Bildschirm. Eine einfache FOR- oder auch WHILE-Schleife genügt, um das Bild zu bewegen. Die Schleife ändert ganz einfach nur die Position der PUT-Grafik, und jedesmal wird das Bild neu gezeichnet. Das ist sicherlich die einfachste Methode. Dabei müssen Sie aber Folgendes beachten: Verwenden Sie beim PUT-Befehl das Aktionsverb PSET, denn sonst ergibt sich ein Mischmasch auf dem Bildschirm, das garantiert niemand erkennt.

Eine andere Möglichkeit wäre, jedesmal den Bildschirm zu löschen, bevor die nächste Phase des Bildes auf den Bildschirm plaziert

wird. Das flimmert aber leider fürchterlich. Aber auch hier können wir wieder alle Finten anwenden, die wir schon in der Abhandlung über Animation mit dem PUT-Befehl gelernt haben. So können wir ja bereits ein Bild im Hintergrund haben und das neue darüberscrollen, indem wir das Bild zweimal mit Exklusiv-Oder an dieselbe Stelle bringen und dann erst weiterbewegen. Damit hätten wir per Software eine Fähigkeit emuliert, die der Amiga eigentlich hat, die uns das BASIC nur nicht zugänglich macht: Das Dual-Playfield-Scrolling. Auch hier flimmert es wieder ein klein wenig. Wann und wie stark das Flimmern ist, hängt auch von der Anzahl der Bitplanes und der Farben ab. Hier hilft also wieder nur die Methode: Ausprobieren, bis die beste Alternative gefunden ist.

Das Scrolling eines Bildes mit dem PUT-Befehl geht schnell und zuverlässig, wenn es auch nicht immer sehr schön aussieht. Und sogar eine gleichzeitige Animation im Rest Ihres Programmes ist möglich. Verwenden Sie dazu einfach das Event-Trapping des Timers mit ON TIMER GOSUB. Dadurch wird das Scrolling und auch das restliche, »gleichzeitig« laufende Programm langsamer, denn der BASIC-Interpreter muß immer zwischen dem Hauptprogramm und der Scroll-Routine hin- und herspringen. Manchmal fängt es dann allerdings furchtbar zu rucken an. Das liegt daran, daß das Event-Trapping des BASIC nicht unbedingt das zuverlässigste ist. Übrigens gilt das gleiche für die Kollisionsabfrage bei BOBs, die sich manchmal einbildet, daß gar keine Kollisionen stattgefunden haben.

Für das Scrolling eines ganzen, den Bildschirm füllenden Bildes gibt es noch zwei weitere Möglichkeiten, nennen wir sie mal Nummer 2 und 3. Sagen wir lieber, es gibt nur eineinhalb Arten, denn so ganz »astrein« ist Methode 3 nicht.

Methode 2 ist die Anwendung der GET- und PUT-Befehle, wie wir sie bereits im Kapitel über Scrolling hatten. Dabei verwenden wir eine Kombination des SCROLL-Befehls und des PUT-Befehls. Hier darf kein Programm nebenher mitlaufen. Dafür aber haben wir das stufenloseste und flimmerfreieste Scrolling, das in BASIC nur irgend möglich ist. Und so wird es gemacht: Laden Sie das mit dem Umwandler gespeicherte IFF-Bild mit dem Ladeprogramm ein. Ist das erledigt, löschen Sie das Programm. Aber nicht mit NEW, denn so würde auch der Screen mit dem geladenen Bild wieder gelöscht werden, was nicht unsere Absicht ist. Tippen Sie stattdessen im List-

Window ein kleines Programm ein, das sich die Grafik in kleinen Stücken mit GET aus dem anderen Screen holt.

Schalten Sie dazu WINDOW 2 auf das Window mit der Grafik um. Holen Sie dann die Grafik häppchenweise in die Elemente einer Feldvariablen. Wenn wir links oder rechts scrollen wollen, sollten die GET-Stückchen die Grafik von links nach rechts (in senkrechte Streifen) aufteilen. Ich schlage vor, Sie teilen das Bild in Fünftel auf, die Sie der Reihe nach von links nach rechts mit GET in Variablenfelder lesen. Soll von oben nach unten oder von unten nach oben gescrollt werden, fünfteln Sie das Bild von oben nach unten. Beim Entnehmen der einzelnen Grafikstückchen mit GET sollten Sie auch an die Möglichkeit denken, indizierte Variablen zu benutzen. Die können Sie nämlich nachher besser in FOR-Schleifen verwenden. Die so aus der Grafik geholten Stückchen sollten Sie nun in einzelnen sequentiellen Dateien abspeichern, um sie später von anderen Programmen laden zu können.

Nun folgt das Scrolling selbst. Wenn Sie von links nach rechts scrollen wollen, setzen Sie zuerst links das erste Fünftel der Grafik mit PUT in das Fenster. Anschließend rufen Sie so oft den Scroll-Befehl auf, wie das Grafikstückchen breit ist. Ist der SCROLL-Befehl so oft ausgeführt, befindet sich das erste Bildfünftel ein Bildschirmfünftel weiter rechts. Jetzt können wir den nächsten Teil des Bildes wieder mit PUT links anfügen und anschließend wieder das gesamte Bild pixelweise mit dem SCROLL-Befehl um eine Fünftel Bildbreite nach rechts versetzen. Das wiederholen Sie nun für jeden der restlichen Bildstreifen und das Bild ist komplett.

So erscheint immer links ruckhaft ein Bildteil, der Rest jedoch scrollt flimmerfrei von links nach rechts. Das stufenweise Auftreten der Grafik können wir wieder mit der Verdeckungsmethode (indem wie ein Fenster darüber plazieren) vertuschen. Sollte Ihnen der überdeckte Platz zu groß sein, vergrößern Sie ganz einfach die Zahl der Grafikstückchen, um somit den überdeckten Teil zu verringern.

Wie bereits gesagt, ist es sehr schwierig, diese Methode in Programmen gleichzeitig mit anderen Routinen zu verwenden. Verwenden Sie aber neben dieser Scroll-Routine nur Sprites, ist so etwas möglich. Sie erinnern sich noch des Wissens aus dem Kapitel über die OBJECT-Befehle: Sprites und BOBs bewegen sich – solange keine Kollision auftritt – unabhängig vom Rest des Programmes. Zu den BOBs muß

hier aber noch gesagt werden, daß sie einen Störfaktor für das Scrolling darstellen. BOBs werden nämlich softwaremäßig erzeugt und in die Hintergrundgrafik einkopiert. Das bedeutet, daß sie mitscrollen und eine häßliche Spur in der Grafik hinterlassen. Verwenden Sie also zur gleichzeitigen Animation von Hintergrund und Objekten immer Sprites und keine BOBs!

Nun kommen wir zu Methode 3. Sie erlaubt uns, das Bild völlig unabhängig vom Rest des Programmes zu bewegen, ohne auch nur einen Finger dafür krümmen zu müssen. Wir nehmen nämlich das gesamte Bild als BOB! Doch bevor der große Jubel beginnt und die Frage gestellt wird »Warum hat er das nicht gleich gesagt?« - ein Wort der Beschwichtigung. Es geht »nicht ganz.« Das soll heißen, daß ein ganzer Bildschirm zu groß ist, um als BOB verwendet zu werden!

Dabei hatte ich doch gesagt, daß ein BOB beliebig groß werden darf. Das stimmt auch, trifft aber leider nicht auf BASIC zu. Wie Sie wissen (oder nach Lektüre der bisherigen Kapitel wissen sollten), wird ein BOB in einem String abgelegt. AmigaBASIC kann Strings verarbeiten, die eine maximale Größe von 32767 Bytes haben. Und damit kein LoRes-Bild in 32 Farben darstellen, denn 320mal 200 Bildpunkte mal 5 Bitplanes gibt 40 000 Bytes. Ein LoRes-Bild mit 16 Farben, also vier Bitplanes, bekommen wir gerade noch in den String hinein. Wir müssen uns also in der Wahl der Auflösung und der Farben beschränken. Der Rest geht wie von allein: Wir wandeln das Bild in einen BOB-String um, rufen die notwendigen OBJECT-Befehle für die Bewegung auf, und schon bewegt sich der Hintergrund wie von allein. Sie müssen allerdings die Kollisionserkennung zwischen BOB und Fensterrand ausschalten, da sonst das BOB stehen bleibt.

Mit dieser Methode können wir sogar den »Dual-Playfield«-Modus simulieren, der von BASIC unerreichbar ist. Und dabei sind wir nicht einmal auf 3 Bitplanes pro Playfield beschränkt (wie es die Hardware des Amiga ist). Dafür aber müssen wir gegenüber der hardwareunterstützten Fähigkeit einen deutlichen Geschwindigkeitsverlust in Kauf nehmen.

13.4.2 Die BOBs kommen!

Nun wollen Sie sicher erst einmal wissen, wie Sie ein Bild überhaupt zum BOB umwandeln können. Dazu muß man das Format eines BOB-Strings verstehen, das im Listing des Objekteditors (im BASIC-Demos-Ordner unter dem Namen »ObjEdit«) sehr gut dokumentiert ist. (Bitte sehen Sie sich das Format der BOBs aber auch selbst einmal im Objekteditor an. Ich will hier nun wirklich nicht alles nochmal bringen, was jeder Amiga-Besitzer schon hat.) Ich habe das folgende kleine Programm geschrieben, das eine mit GET aus dem Bild geholte Grafik in einen auf Diskette gespeicherten BOB-String verwandelt, den man dann ganz einfach mit *OBJECT.SHAPE Objektnummer,(LOF(Filenummer),Filenummer)* einlesen kann. Dieses Programm müssen Sie allerdings etwas anders starten als bisher gewohnt. Wir wollen nämlich die Variablen und die Grafik des Ladeprogramms nicht löschen. Sehen Sie sich aber zunächst einmal das Listing an.

```
REM BobSaver mit CHAIN aufrufen (siehe Text)
WINDOW 1
PRINT »RETURN druecken und Ausschnitt holen!"
GOSUB Antwort
WINDOW 2
PUT (0,0),pic%,PSET

Checkmouse:
 IF MOUSE(0)<=0 THEN Checkmouse
 left%=MOUSE(3):up%=MOUSE(4):t%=0

click:
 PUT (0,0),pic%,PSET
 LINE (left%,up%)-(MOUSE(1),MOUSE(2)),t%,b
 t%=t%+1:IF t%=2^pic%(2) THEN t%=0
 IF MOUSE(0)=0 THEN click
 right%=MOUSE(3):down%=mouse(4)
 IF right%<left% THEN SWAP right%,left%
 IF down%<up% THEN SWAP down%,up%
```

```
savechecker:
 WINDOW 1
 PRINT "RETURN = SAVE, SPACE = Nochmal"
 GOSUB Antwort
 IF Antwort$=CHR$(13) THEN
   GOTO Speichern
 ELSE WINDOW 2:t%=MOUSE(0):t%=0:GOTO Checkmouse
 END IF

speichern:
 INPUT "Filename ",x$
 WINDOW 2
 PUT (0,0),pic%,PSET
 Planes%=pic%(2)
 ERASE pic%
 t%=3+INT((right%-left%+16)/16)*(down%-up%+1)*Planes%
 DIM bob%(t%)
 GET (left%,up%)-(right%,down%),bob%
 OPEN x$ FOR OUTPUT AS 1
 PRINT#1,MKL$(0);MKL$(0);MKI$(0);
 PRINT#1,MKI$(Planes%);MKI$(0);
 PRINT#1,MKI$(bob%(0));MKI$(0);
 PRINT#1,MKI$(bob%(1));MKI$(24);
 PRINT#1,MKI$(2^Planes%-1);MKI$(0);
  FOR t%=0 TO UBOUND(bob%)-1
   PRINT#1,MKI$(bob%(t%));
  NEXT
 CLOSE 1
END
```

Um diese Routine zu benutzen, müssen Sie zunächst ein mit dem IFF-Wandler gespeichertes Bild mit dem Ladeprogramm aus dem Beispiele-Kapitel einladen. Ist das Programm beendet, können Sie mit Amiga-M und Amiga-N (linke Amiga-Taste) zwischen Bild und Output-Window hin- und herschalten. Klicken Sie in das BASIC-Output-Window und geben Sie jetzt den folgenden Befehl ein.

```
CHAIN "Bobsaver",,ALL
```

Damit bleiben alle Variablen und die Grafik des Ladeprogrammes erhalten. Der »Bobsaver« arbeitet damit. Ist das Programm gestartet, drücken Sie die RETURN-Taste. Klicken Sie jetzt mit der

Maus in die linke obere Ecke des Bildteiles, das Sie als BOB benutzen wollen. Nun können Sie mit der Maus ein Rechteck bis zum unteren rechten Ende des BOBs ziehen. Klicken Sie nochmal auf den linken Mausknopf. Das Programm fragt nach einem Filenamen und speichert den gewählten Bildteil als BOB ab.

Damit können Sie sich nun alle Objekte, die Sie für BASIC-Programme benutzen wollen, mit einem professionellen Grafikprogramm erstellen. Sie können kleine BOBs verwenden, Sie können aber auch ein ganzes Bild verwenden – sofern es weniger als 32 767 Bytes Speicherplatz benötigt.

Damit steht nun auch einer gleichzeitigen Animation vieler Objekte nichts mehr im Weg. Denken Sie bei der Programmierung aber auch immer daran, daß die Angabe der Bewegung durch die OBJECT-Befehle nicht genügt. Jede Bewegung eines Objekts stoppt zunächst einmal bei einer Kollision mit einem anderen Objekt. Setzen Sie deshalb die Kollisionsmasken der einzelnen Objekte so fest, daß nicht immer gleich alles mit jedem kollidiert und Ihr Programm sich um zuviel Kollisionsabfragen kümmern muß. Soll zum Beispiel ein ganzes Bild als BOB einen scrollenden Hintergrund bilden, so sollte man gleich von vornherein sämtliche Kollisionsmöglichkeiten dieses Objektes ausschalten.

Damit ist eigentlich schon alles erledigt, was bei der gleichzeitigen Animation von Objekten und Hintergrund zu tun ist. Wie Sie dieses Wissen in eigene Projekte einfließen lassen, liegt bei Ihnen. Wichtig ist, daß die Objekte nicht wild durch die Gegend fliegen, sondern Ihr Programm die Kontrolle darüber hat. Da die Anzahl der möglichen Anwendungen der beschriebenen Techniken so groß ist, kann ich Ihnen unmöglich sagen, wie das genau geschieht.

Wenn Sie in diesem Buch nun aber schon so weit gekommen sind und auch alle Übungen praktisch mitgemacht haben, werden Sie genug Erfahrung mit AmigaBASIC haben, um selbst die Kontrolle herbeiführen zu können. Greifen wir trotzdem ganz kurz ein Beispiel heraus: ein Ballspiel der guten alten »Space-Invaders«-Sorte. Die Aliens bewegen sich von selbst – also verwenden Sie BOBs. Immer wenn ein Alien an den rechten oder linken Rand des Fensters stößt, müssen Sie seine Position eine Zeile tiefer setzen und die Bewegungsrichtung von links nach rechts oder umgekehrt ändern. Die eigene Abschußrampe bleibt unter ständiger Kontrolle des Spielers.

Dafür empfehle ich ein Sprite, dessen Position nur mit OBJECT.X oder OBJECT.Y geändert wird, oder eine in ein GET-Array abgelegte Abschußbasis. Der eigene Schuß kann ein mit OBJECT.START bewegtes Sprite sein. Und hinter den Aliens ist natürlich noch Platz für einen unbewegten Sternenhintergrund. Den Rest überlasse ich einmal Ihnen und Ihren Fähigkeiten.

13.4.3 Dreidimensionale Bewegungen

Das Programm »3D-Grafik« aus dem Beispielteil haben Sie sicher schon eingetippt und ausprobiert. Natürlich werden Sie sich fragen, wie Sie solche Grafiken in eigenen Programmen einbauen können – nur in Realzeit berechnet und dreidimensional animiert. Die Antwort darauf ist ganz eindeutig: Lernen Sie eine bessere Programmiersprache, beherrschen Sie das Amiga-Betriebssystem und legen Sie los. Als Grundlage dazu kann Ihnen vielleicht die kleine Einführung im nächsten Buchteil dienen.

Doch irgendwie wollen wir doch diese tolle Möglichkeit der 3D-Grafik verwerten. Sie werden sich schon denken können, daß ich gleich mit mehreren Ideen dazu herausplatzen werde. Wenn Sie ein wenig überlegen, fällt Ihnen bestimmt auch selbst etwas zu diesem Thema ein. Da gibt es zum Beispiel den GET- und den PUT-Befehl, mit denen wir aufeinanderfolgende Bildsequenzen erscheinen lassen können. Da gibt es die Möglichkeit, die von unserem Programm berechneten Realkoordinaten in DATA-Zeilen zwischenzuspeichern und wiederzuverwenden. Und noch weitere unzählige Verwendungsmöglichkeiten der verschiedenen Befehle, so zum Beispiel die BOBs, in die wir diese 3D-Grafiken ja ebenfalls einspeichern können.

Gehen wir einige Variationen durch: Die GET- und PUT-Methode ist Ihnen sicher klar. Sie brauchen dazu nur im 3D-Programm nach dem Darstellen der Grafik ein GOSUB einzufügen, das zur eigenen Routine springt. Die Routine holt sich das Bild mit GET und speichert es dann ab. Nach dem Speichern des Objektes verändern Sie den Blickwinkel um ein paar wenige Grade, lassen dieses Bild berechnen, speichern es in einer anderen Datei und so weiter. Damit haben Sie alle Animationssequenzen, die Sie benötigen, gespeichert. Die Methode zum Darstellen einer so animierten 3D-Grafik haben wir bereits mit unserem kleinen Beispiel-Listing aus dem Kapitel über Animation mit PUT besprochen. Wenn Sie beim Wechseln der

verschiedenen Arrays (oder der Indizes eines mehrdimensionalen Arrays) auch noch die Startkoordinaten im PUT-Befehl verändern, kann sich so beispielsweise ein 3D-Würfel drehend übers Bild bewegen.

Eine weitere Möglichkeit ist die Speicherung der Realkoordinaten. Unser 3D-Programm berechnet anhand der Koordinaten eines 3D-Punktes die Realkoordinaten »px« und »py« in der Bildschirmebene. Die Koordinaten aller Punkte sind in den Arrays »px(0)« bis »px(Ende)« und »py(0)« bis »py(Ende)« gespeichert. Das Parameter »Ende« bezeichnet die Anzahl der gespeicherten Punkte und wird im Programm als Schleifenzähler »a« in der Routine »Schleife« angegeben.

Alles, was Sie nun tun müssen, ist es, die Koordinaten irgendwie aus dem Programm herauszubekommen, zum Beispiel als sequentielle Datei zu speichern. Danach können Sie jederzeit die Koordinaten des Objektes mit LINE-Befehlen verbinden. Das ist dann zwar genauso langsam wie im 3D-Programm, bringt aber eine noch recht akzeptable Geschwindigkeit. Doch auch hier gibt es wieder eine Alternative. Wenn der Speicher reicht, öffnen Sie zwei Screens. In jedem Screen öffnen Sie ein Window. Lassen Sie dann die drei folgenden Befehle ausführen.

```
WINDOW 1
WINDOW OUTPUT 2
CLS
```

Damit holen Sie Window 1 und damit Screen 1 nach vorn (wenn wir Window 1 in Screen1 und Window 2 in Screen 2 gesetzt haben). Dann wird jedoch die Ausgabe auf den zweiten Screen umgelenkt , der nicht sichtbar ist und dieser dann gelöscht.

Dann müssen Sie das Bild aufbauen. Das geschieht in einer kleinen Schleife, die alle Elemente von »px(0)« und »py(0)« bis »px(Ende)« und »py(Ende)« durchgeht und mit Linien verbindet. Sie sehen diese Linien aber (noch) nicht, da der entsprechende Screen ja unsichtbar ist. Lassen Sie dann die drei folgenden Befehle ausführen.

```
WINDOW 2
WINDOW OUTPUT 1
CLS
```

Damit machen Sie das eben gezeichnete Bild sichtbar (holen den hinteren Screen nach vorn). Sie können nun in dem ehemals vorn – jetzt aber hinten liegenden – Screen 1 die nächste Bewegungsphase zeichnen. Und so weiter. So erzeugt man einen wenigstens teilweise flüssigen Bewegungsablauf, da der Bildaufbau unsichtbar vor sich geht. Die einzelnen Bildsequenzen müssen dazu natürlich vorher in die Felder »px« und »py« eingelesen werden. Die beste Methode ist hier, entweder vor dem Start der Animation sämtliche Daten von Diskette einzulesen oder diese Zahlen in DATA-Zeilen abzulegen.

Ich empfehle außerdem, indizierte Variablen zu verwenden, also beispielsweise px(1,2) für die x-Koordinate des zweiten Punktes (Arrays fangen normalerweise beim Element Null an!) der dritten Bewegungsphase, py(3,1) für die vierte y-Koordinate der zweiten Bewegungsphase und so weiter. Der Vorteil der doppelt indizierten Feldvariablen liegt hier nicht nur in der einfacheren Abarbeitung innerhalb von FOR-Schleifen. So können Sie nämlich auch die Bewegung besser steuern. Wenn ein Spieler zum Beispiel den Joystick nach oben bewegt, kann Ihr Programm darauf reagieren, und braucht dazu nur den Variablenfeld-Indiz für die Bewegungsphase des Objektes um 1 zu erhöhen. Bewegt der Spieler den Joystick nach unten, erniedrigen Sie den entsprechenden Wert um 1. So bewegt sich das Objekt dreidimensional um den gewünschten Winkel, wenn der Spieler den Joystick nach vorn drückt, und dreht sich zurück, wenn er ihn auf sich zu zieht. So einfach kann man dann eine durch Eingabegeräte gesteuerte dreidimensionale Bewegung anzeigen.

13.5 Das Ende

Damit wären wir am Ende des kleinen BASIC-Kurses angelangt. Doch zum Schluß folgt noch eine Unverschämtheit meinerseits: Setzen Sie sich nicht auf die faule Haut! Erst die Programmierpraxis bringt die gewünschten Ergebnisse, und auch jemand, der bereits Jahre Programmiererfahrung in den verschiedensten Sprachen auf dem Buckel hat, kann sich oft schwertun, wenn er nicht die nötige Praxis hat. Ich kann Ihnen deshalb nur raten: Üben – Üben – Üben.

Auch wenn BASIC Ihnen nicht zusagt – zum Ausprobieren bestimmter Algorithmen ist so ein Interpreter ganz gut. Ich programmiere inzwischen zwar auch in anderen Sprachen, verwende BASIC aber immer noch des öfteren zum schnellen Ausprobieren bestimmter Tech-

niken. Doch nun für die, die in höhere Gefilde streben, der nächste Buchteil! Lernen sie die Tiefen (und Untiefen) der Systemsoftware kennen!

Buchteil 4

Dieser Buchteil ist eine Enführung in das Betriebssystem des Amiga. Diese Einführung soll dem Einsteiger erleichtern, die komplexen Systeme, die hinter der Systemsoftware stecken, zu verstehen. Wer wirklich professionell programmieren will, muß entweder alles selbst machen, oder auf die Resourcen der Betriebssystemsoftware zurückgreifen. Die wichtigsten Teile des Betriebssystems für unsere Zwecke sind:

Die Intuition-Funktionen

Die Grafiklibraries

Die Animationsbibliotheken

Die Layerbibliothek

Die Textfunktionen

Mit diesem Teil des Buches schaffen Sie sich einen Grundstock an Wissen über die Grafiksoftware des Amiga. Systemprogrammierer werden aber vergeblich nach Beschreibung der Betriebssystemfunktionen suchen.



14

Das Amiga-Betriebssystem

14.1 Einführung

In der einschlägigen Literatur heißt es, das Betriebssystem des Amiga sei »AmigaDOS«. Aber, so muß dann der hoffnungsvolle Jungprogrammierer erfahren, die Benutzeroberfläche, mit der er am Amiga arbeitet, heißt »Intuition«! Im Handbuch und auf dem Bildschirm erfährt er aber, das Programm hieße »Workbench« und in einem Fachmagazin erfährt er, die Programme liefen unter »Exec«, die Grafik- und Soundfunktionen wären »Libraries«, die man mit einer Exec-Funktion aktivieren könne, welche wiederum das DOS zum Laden benutze und so weiter und so fort. Tja, das waren noch Zeiten, als man einfach sagen konnte, das ist mein C64- oder Apple-Betriebssystem und ich springe bestimmte Betriebssystemroutinen einfach mit ihrer Adresse an.

Nach meiner Aktion »Licht ins Dunkel« in den folgenden Abschnitten wird dem Amiga-Besitzer hoffentlich etwas klarer werden, welche Systemsoftware sich in seinem Computer breitmacht – dieses Grundwissen braucht man einfach für die vernünftige Programmierung des Amiga.

14.2 Ausflug ins Betriebssystem

Zu Beginn unserer kleinen Reise in die große Welt des Amiga-Kernels und speziell seiner Grafikroutinen ist es erst einmal wichtig zu wissen, wie die Teile der Systemsoftware zusammenhängen. Wir haben es hier nicht nur mit einem einzigen System zu tun, sondern

vielmehr mit einer Kombination von vielen Systemen. Jedes dieser Systeme hat seine eigenen speziellen Aufgaben und wurde (fast) unabhängig von den restlichen Systemen entwickelt. Die Entwickler der einzelnen Teilsysteme arbeiteten dabei allerdings zusammen, so daß alle Teile miteinander arbeiten können. Lediglich das Amiga-DOS wurde dem Betriebssystem aufgepfropft. Man baute das System »Tripos«, das auch auf einigen Minicomputern läuft, auf den Amiga um und versuchte, es an die anderen Teilsysteme anzupassen. In Betriebssystemversion 1.0 hatten die Softwareentwickler heftig mit den sich daraus ergebenden Problemem zu kämpfen, doch mit der neuesten Version 1.2 hat sich AmigaDOS fugenlos in die Gesamtheit eingeordnet.

Jedes Teilsystem der Amiga-Systemsoftware ist eine Bibliothek von Funktionen, eine »Library«. Diese Libraries oder Bibliotheken sind in einer sogenannten »Soft-Architektur« aufgebaut. Dazu aber später noch mehr. Die einzelnen Komponenten des Amiga-Kernels sind die ROM-Libraries, die Disklibraries, »Devices« und »Resources« sowie die »Linker Libraries«. Das sagt Ihnen wahrscheinlich im Augenblick gar nichts. Deswegen folgen nun ein paar knappe Erklärungen zu den einzelnen Teilen des Betriebssystems und deren Bedeutung.

14.2.1 ROM-Libraries

Die ROM-Libraries werden von der Kickstart-Diskette in das schreibgeschützte RAM gelesen. Die Datenstrukturen, auf denen diese Libraries aufbauen, werden beim Booten initialisiert.

EXEC: Dem gesamten System liegt das »Exec« zugrunde, das heißt alle anderen Teile des Systems können erst durch Exec aktiviert werden. Exec ist, wie der Name schon sagt, die ausführende Ebene (Executive). Es steuert den Zugriff auf andere Libraries und deren Ausführung. Exec kontrolliert Tasks und Interrupts, sogenannte »Messages« (für den Datenaustausch) und sorgt für die Speicherverwaltung sowie die Listenverwaltung. Listen bilden die Grundlage für das Zusammenarbeiten der einzelnen Libraries. Alle Betriebssystemteile verständigen sich untereinander durch diese Listen und sind aufbauend auf diese Listen entwickelt worden. Exec ist die niedrigste und hardwarenäheste Ebene des Amiga-Betriebssystems.

GRAFIK: Die »Graphics Library«, auf die wir in Kapitel 3 näher eingehen werden, hat zwei Hauptaufgaben: Sie stellt dem Programmierer einmal eine direkte Schnittstelle zur Grafikhardware zur Verfügung (Bildanzeigeprimitive wie Views, Viewports, Bitmaps sowie direkter Zugriff auf Copper, Blitter und Spritelogik) und bietet Grafikprimitive zum Zeichnen von Linien, zum Füllen von Flächen und so weiter. Teil der Grafikbibliothek sind auch die »Bibliotheken in der Bibliothek« für Text und GELs (*Graphics Elements*: Sprites, VSprites, Bobs, Animobs).

LAYERS: Routinen, die mit der Grafiklibrary zusammenarbeiten, um den Bildschirm wie eine Ansammlung übereinandergelegter »Schichten« behandeln zu können. Diese Layers bilden die Grundlage für die Fensterverwaltung des Amiga. Die Routinen der Layers Library »kümmern« sich auch um das Auffrischen gelöschter Teile eines Layers, die Zwischenpufferung von überdeckten Layers, Größenänderungen und so weiter.

INTUITION: Intuition ist eine Bibliothek, deren Funktionen die Routinen der Grafiklibrary und der Layers Library verwenden. Intuition ist die Standard-Benutzeroberfläche des Amiga. Intuition kümmert sich um viele Dinge, die man am Bildschirm sieht, wie Fenster, Maus und Mauszeiger, Pull-Down-Menüs, Gadgets, Requester und die Ein/Ausgabe. Natürlich können alle diese Dinge über die Intuition-Funktionen auch vom Programmierer aufgerufen, verwendet und verändert werden. Intuition bietet außerdem Möglichkeiten, die etwas komplizierten Routinen der Grafiklibrary zu umgehen. Ein gutes Beispiel dazu sind die Screens. Hier brauchen nur die Parameter wie Auflösung, Größe und so weiter in eine Datenstruktur geschrieben werden und anschließend die Routine *OpenScreen* mit einem Zeiger auf diese Struktur aufgerufen werden – fertig. Wir brauchen uns also nicht um das Reservieren von Speicherbereichen und dergleichen kümmern.

CLIST: Der Name dieser Bibliothek steht für »Character List«. Diese Library ist für das String-Handling, also die Verwaltung von Strings im Speicher, zuständig. CLIST verwaltet den Stringspeicher, vollführt Längen- und Index-Operationen, Blocksatz und Stringänderungen. CLIST benutzt – wie alles im Amiga – Listen zur Stringverwaltung. Wollen wir Text in unserem Grafikschirm darstellen, brauchen wir die CLIST-Funktionen des Öfteren.

MATHFFP: Der Name dieser Bibliothek steht für »Mathematics/Fast Floating Point«. Sie enthält Routinen für besonders schnelle Ausführung von Addition, Subtraktion, Multiplikation, Division, Absolutwertberechnung, Negation, Integer-Konversion, Nulltest und Vergleich bei niedriger Genauigkeit. Schnelle Mathematikfunktionen sind insbesondere im Bereich 3D-Grafik-Berechnungen von großer Wichtigkeit; Realzeitanimation benötigt eben schnelle Mathematik.

DOS: In dieser Bibliothek sind die AmigaDOS-Funktionen *Open, Close, Read, Write, Protect* und so weiter zu finden. AmigaDOS regelt die Disketten-Ein/Ausgabe sowie die Kontrolle der Multitasking-Prozesse, die von Exec ausgeführt werden (Alles klar? Hier noch einmal eine kleine Zusammenfassung: Exec führt die Tasks durch Aufteilung der Prozessorzeit aus, AmigaDOS kontrolliert sie und wacht über ihre Prioritäten, und Intuition verwaltet die verschiedenen Windows, in denen die Tasks laufen).

RAM-LIB: Diese Bibliothek dient zur Verwaltung der RAM-Disk.

14.2.2 Disklibraries

Diese Bibliotheken befinden sich auf der Workbench-Diskette und werden nur dann in den RAM-Speicher gelesen (und mit Exec-Routinen in das System eingebunden), wenn sie benötigt werden. Immer, wenn eine dieser Bibliotheken von einem Programm benötigt wird, aber gerade nicht im Speicher ist, wird sie vom DOS aus dem LIBS-Directory geladen.

ICON: Diese Bibliothek enthält Routinen, die von der Workbench benutzt werden, um Speicher für Workbench-Objekte zu reservieren beziehungsweise freizusetzen. »Workbenchobjekte« sind die Icons (Schubladen, Tools, Projekt-Icons, der Mülleimer und so weiter). ICON steuert außerdem den Update von Filenamen (»Copy of«) und das korrekte Setzen der entsprechenden Info-Files beim Kopieren eines Programms. Der CLI-Befehl »LoadWb« tut nichts anderes als Intuition zu aktivieren, die ICON-Routinen zu laden, und diese dann zu starten. Die »Workbench« ist somit kein Betriebssystemteil, sondern nur die logische Konsequenz der Zusammenarbeit der Betriebssystemteile Intuition und Icon.

MATHTRANS: Diese Bibliothek enthält Routinen für trigonometrische Funktionen auf Basis der Fast-Floating-Point-Library (siehe oben), sowie Funktionen zur Umwandlung zwischen dem IEEE-Standard-Double-Precision-Format und Fast-Floating-Point-Format.

MATHIEEEDOUBBAS: Diese Bibliothek enthält die gleichen Routinen wie die MATHFFP-Library. Nur sind diese nicht so schnell, aber dafür doppelt so genau und halten das Datenformat nach dem IEEE-Standard ein.

TRANSLATOR: Dies ist die einzige Funktions-»Bibliothek« mit nur einer einzigen Routine. Diese Routine *Translate* wandelt englischsprachige Texte in Phoneme um, die der sogenannte »Narrator« dann in Sprachausgabe umsetzen kann.

DISKFONT: Diese Bibliothek enthält eine Liste aller Fonts (Zeichensätze), die gerade im Speicher oder auf der Diskette sind. DISKFONT wird nur dann einen vom Programm angesprochenen Zeichensatz von der Diskette laden, wenn er nicht schon im Speicher ist. Der Programmierer braucht sich also nicht mehr selbst darum kümmern, ob ein Zeichensatz im Speicher ist oder nicht.

Das sind derzeit alle Standard-Libraries, die in der aktuellen System-Dokumentation erwähnt werden. Die neueren Betriebssystemversionen enthalten im LIBS-Directory allerdings noch einige Libraries mehr (VERSION, INFO) über die mir keine Informationen zur Verfügung stehen.

14.2.3 Devices und Resources

Neben den Standard-Bibliotheken gibt es noch die »Devices« und »Resources«, die eine etwas andere Art von Bibliotheken darstellen. Die »Devices« sind Routinen (oder in einigen Fällen besser ausgedrückt Gerätetreiber), die von EXEC-Routinen wie *DoIO* benutzt werden. Die Devices erlauben hardwareunabhängige Ein-/Ausgabe-Funktionen wie *Reset, Read, Write, Update* und *Clear*. (Der Fachmann sagt »Gerätetreiber« zu solchen Bibliotheken). Wie normale Libraries können Devices speicherresident sein oder bei Bedarf von der Diskette geladen werden. Standardmäßig resident sind die folgenden Devices: *Timer, Trackdisk, Keyboard, Gameport, Input, Console* und *Audio*. Von Diskette geladene Devices (sie liegen im DEVS-Directory) sind: *Narrator, Serial, Parallel, Printer* und *Clip-*

board. Das »Clipboard« ist eine interessante Alternative zum Zwischenspeichern von Texten und Grafiken, wenn solche Daten zwischen Programmen oder Tasks ausgetauscht werden sollen. Auf dem Apple Macintosh ist das Clipboard-Format der wichtigste Standard zum Datenaustausch. Von den Amiga-Software-Entwicklern wird das Clipboard leider meist nicht genutzt. Dafür verfügt auf dem Amiga jedes bessere Programm über einen IFF-Format-Lader.

Eine »Resource« ist eine Library ohne indirekte Einsprungadressen (zu den indirekten Einsprungadressen später mehr). Resources sind sehr eng an die Hardware gebunden. Mit den Resource-Routinen kann direkter Einfluß auf die Amiga-Hardware ausgeübt werden. Die Diskettenlaufwerke, die zwei CIAs (Ein-/Ausgabe-Chips), das POTGO-Register und die Registerbits des seriellen und parallelen Ports können zum Beispiel auf diesem Wege manipuliert werden. Die Funktion der einzelnen Resource-Routinen ist es, die Hardware zu kontrollieren, indem bestimmten Tasks der exklusive Zugriff auf bestimmte Hardwarekomponenten erlaubt oder verboten wird. Normalerweise werden diese Aufgaben von Software höherer Ebene (Intuition, Grafiklibrary, Devices) erledigt. Wer jedoch direkten Einfluß auf die Hardware nehmen will, kann dies mit den Resources tun. Der Blitter ist übrigens keine »Resource«, er wird direkt von der Grafiklibrary mit Routinen wie *OwnBlit* kontrolliert.

14.2.4 Linker-Libraries

Linker-Libraries sind Bibliotheken, die nachträglich geladen werden können. Die Bibliothek »Amiga.lib« zum Beispiel enthält unter anderem Standard-Funktionen der Programmiersprache C, wie zum Beispiel *printf*. Jeder Programmierer kann sich aber auch eigene Routinen schreiben und sie mit Hilfe des Linkers ALINK in eigene oder bereits vorhande Linker-Libraries legen.

14.3 Die weiche Architektur – wie man das System anspricht

Nun sollten Sie wissen, aus wieviel scheinbar unbedeutenden Kleinteilen das Betriebssystem besteht. Abbildung 14.1 zeigt noch einmal,

wie die wichtigsten Teile des Systems zusammenarbeiten. Wenn man sagt, Software liefe »unter AmigaDOS«, meint man also, das Programm verwendet die AmigaDOS-Routinen. Jedes Amiga-Programm läuft aber unter Exec und muß nicht unbedingt Intuition benutzen.

Das Amiga-Kernel ist nun aber nicht, wie etwa beim guten alten C64, eine Ansammlung von Routinen, die hintereinander an einer bestimmten Stellen im Speicher liegen. So etwas würde man eine »harte Software-Architektur« nennen. Wir haben es beim Amiga aber mit einer sogenannten »Soft Machine Architecture« zu tun. Das bedeutet, daß jede Library verschiebbar ist, also überall im Speicher liegen kann. So ist das System wesentlich flexibler, da eine Bibliothek immer in den gerade freien Raum geladen werden kann. Zudem garantiert dieses Konzept Aufwärtskompatibilität. Commodore kann eine neue Version des Betriebssystems herausbringen, in dem alle Routinen an anderen Stellen sitzen, und trotzdem funktioniert alle Software so wie vorher.

Der Nachteil dieses Konzeptes liegt in der geringeren Geschwindigkeit. Wenn man nicht mit absoluten Adressen arbeiten kann, sondern sich mit indirekter Adressierung herumschlagen muß, werden Systemaufrufe viel langsamer. Daß der Amiga trotz der langsamen Systemsoftware nach wie vor sehr schnell ist, hat er seinen Custom-Chips und DMA-Kanälen zu verdanken. (Ein weiterer Grund für das nicht gerade schnelle Betriebssystem liegt darin, daß es in C geschrieben wurde.)

Wie funktioniert nun diese Soft-Architektur? Wie springen wir Betriebssystemroutinen an, wenn wir nicht wissen, wo sie liegen? Ist eine Bibliothek erst einmal geladen, müssen wir zunächst ihre Basisadresse herausfinden. Alle Routinen einer Library beginnen bei negativen Offsets von dieser Basisadresse aus. In der Bibliothek Intuition sieht das zum Beispiel so aus: die *Open*-Funktion liegt bei (-6) von dieser Basisadresse, *DrawBorder* bei (-108), *DrawImage* bei (-114) und so weiter.

Die Basisadresse einer Bibliothek finden wir, indem wir die Exec-Routine *OpenLibrary* aufrufen. Ach, Sie wissen nicht, wo diese ist? Tja, da müssen wir wohl erst die Basisadresse von Exec herausfinden. Die steht aber glücklicherweise immer in der selben Adresse: Adresse 4 ist die einzige absolute Adresse des Betriebssystems und enthält einen Zeiger auf die Exec-Basisadresse *ExecBase*. Von dieser

Adresse aus nehmen wir den Offset zur OpenLibrary-Funktion, rufen damit Intuition auf, erhalten von diesem Aufruf einen Zeiger auf die Intuition-Basis *IntuitionBase*, nehmen dann den Offset zur gewünschten Intuition-Routine – und sind endlich dort wo wir hinwollten!

Wie Sie sehen ist die Architektur der Amiga-Software doch ein bißchen komplexer als die Betriebssysteme anderer Computer. Und sollten Sie mal wieder gefragt werden, wie man denn in den Blitter POKEd oder wo die Adresse der Routine zur Textausgabe sei, dann dürfen Sie über Libraries, Resources, Betriebssystemteile und indirekte Adressierung reden. Legen Sie dazu bitte auch das sadistische Grinsen auf, das ich mir zu Anfang gefallen lassen mußte!

Doch nun zu den einzelnen Teilen des Amiga-Systems, die uns als Grafikprogrammierer besonders interessieren. In den folgenden Kapiteln soll beschrieben werden, welche Libraries Sie wofür benötigen.

15

Die Intuition-Funktionen

15.1 Was ist Intuition?

Intuition ist die Schnittstelle zwischen dem Benutzer und dem Computer. Es handelt sich dabei um eine Betriebssystem-Bibliothek, die etwa 60 Funktionen beinhaltet. Diese Funktionen sollen es dem Programmierer ermöglichen, eine Kommunikation (auf grafischer Ebene) zwischen Mensch und Computer so einfach wie möglich zu gestalten. Die Intuition-Funktionen sind für die möglichst einfache Handhabung von Windows, Screens, Gadgets, Menüs und Requestern entwickelt worden.

15.2 Gadgets, Requester und Alerts

Gadgets und Requester sind neben Menüleisten die meistverwendete Methode der grafischen Benutzerführung. Intuition unterscheidet zwischen zwei Typen von Gadgets: den Systemgadgets und den Applikationsgadgets. Die Systemgadgets werden automatisch von Intuition in die Windows oder Screens plaziert und auch von Intuition verwaltet, sofern wir die entsprechenden Bitflags der Screenstrukturen beziehungsweise Windowstrukturen setzen. Die Systemgadgets kann man nicht durch die Intuition-Funktion *RemoveGadget* entfernen. Es gibt vier verschiedene Systemgadgets:

• Das DRAG-Gadget: Es beinhaltet die gesamte Titel-Leiste eines Screens oder Windows und wird vom Benutzer verwendet, um das Window oder den Screen zu verschieben.

• Die DEPTH-Gadgets: Die Depth- oder Tiefen-Gadgets erscheinen in der rechten oberen Ecke eines Screens oder Windows. Sie erlauben es dem Benutzer, die Reihenfolge der Screens und Windows, also welche Screens oder Windows im Vorder- beziehungsweise Hintergrund liegen, zu verändern.

• Das SIZING-Gadget: Es liegt in der rechten unteren Ecke eines Windows und kann dazu verwendet werden, die Größe eines Windows zu ändern.

• Das CLOSE-Gadget: Es erscheint in der linken oberen Ecke eines Windows oder Screens. Mit ihm kann das Fenster oder der Screen geschlossen (vom Bildschirm entfernt) werden.

Da sich bereits das Intuition-System der System-Gadgets annimmt, braucht man seine Programme nicht so zu schreiben, daß sie sich um die System-Gadgets kümmern müssen (mit Ausnahme des CLOSE-Gadgets). Das Programm braucht also nicht nach Aktionen des Benutzers, die diese Gadgets betreffen, Ausschau halten und darauf reagieren.

Requesters sind Display-Elemente, die den Benutzer ersuchen (engl. to request = ersuchen), (darauf) mit Anklicken einer der »Requester-Boxen« zu reagieren. Sie können entweder vom System oder von Applikationen angezeigt werden. Requesters müssen vom Benutzer erst beantwortet werden, bevor weiterer Input in dem jeweiligen Window möglich ist. Das erklärt auch den Unterschied zwischen Requestern und Gadgets.

Zusätzlich zu Requestern und Gadgets gibt es noch die sogenannten Alerts (engl. *Alert* = Alarm). Alerts sind Fehlermeldungen, die am oberen Ende des Bildschirms erscheinen, wenn eine extrem kritische Kondition des Systems auftritt. Alerts fallen, wie die Gadgets, in zwei Kategorien: System-Alerts und Applikations-Alerts. Ein Beispiel für einen Alert ist die »Guru-Meditation«, die jedem Amiga-Besitzer ein Begriff sein dürfte. Sie ist ein System-Alert. Man kann aber auch die eigene Applikation so programmieren, daß ein solcher Alert erscheint, wenn eine Situation eintritt, die die Weiterführung des Programms verhindert.

15.3 Ein- und Ausgabe mit Intuition

Die Ein- und Ausgabe in Windows kann einmal direkt über Intuition durch sogenannte »Messages« (Nachrichten) über den IDCMP (siehe unten) vorgenommen werden. Alternativ ist auch ein Mechanismus möglich, der »Console Device« genannt wird.

15.3.1 Eingabe

Es gibt zwei Arten, Informationen über die Eingaben des Benutzers zu erhalten. Die eine Art ist direkt und liefert recht »rohe« Daten ohne vorherige Bearbeitung durch das Betriebssystem. Die zweite Möglichkeit versorgt uns mit allen Standard-Input-Funktionen, automatischer Datenpufferung und ASCII-Daten, wie sie auch von AmigaDOS bearbeitet werden. Diese Möglichkeiten sind:

• Der IDCMP, kurz für *Intuition Direct Communications Message Port*. Über diesen Weg erhält man die Daten roh als Flags, Bits und Bytes. Damit empfängt man beispielsweise, ob die Maus bewegt wurde, ob ein Gadget aktiviert wurde und so weiter.

• Das »Console-Device«. Hierfür öffnen wir einfach ein Window als *Gerät* »CON:« (also wie ein Disk-Drive oder ein Drucker), von dem wir den Input durch die AmigaDOS-Routinen erhalten. So können wir beispielsweise die Tastatureingaben in dem als Console Device eröffneten Window ganz einfach durch das AmigaDOS-Kommando *Read* empfangen. Hier brauchen wir die Daten nicht mehr zu bearbeiten. Alle Daten, die auf diesem Weg an ein Programm gemeldet werden, werden automatisch in Standard-ASCII-Zeichen beziehungsweise ANSI-Escape-Sequenzen umgewandelt. Durch Intuition erzeugte Ereignisse wie zum Beispiel CLOSEWINDOW (das Anklicken des Close-Gadgets) werden ebenfalls in Escape-Sequencen umgewandelt.

15.3.2 Ausgabe

Für die Ausgabe auf dem Bildschirm über Intuition gibt es drei Möglichkeiten:

• Bildschirmausgabe durch direktes Ansprechen der Grafik-, Text- und Animationsroutinen des Betriebssystems. Das ist die »nie-

drigste« Ebene der Bildschirmausgabe, auf der wir beispielsweise Linien ziehen, Flächen füllen, Animation erzeugen und unformatierten Text ausgeben.

• Ausgabe durch Ansprechen der Intuition-Funktionen. Diese arbeiten auf höherer Ebene, sind einfacher zu programmieren als die zuerst genannten Routinen (greifen aber selbst auf diese Routinen zu). Viele der Intuition-Routinen machen zwar dasselbe wie einige der Routinen aus den Grafik- und Textbibliotheken, nehmen aber dem Programmierer die ganze Arbeit des Initialisierens, Zwischenspeicherns und so weiter ab.

• Textausgabe über das Console-Device (siehe oben). Texte können hiermit zum Beispiel mit automatischem intelligenten Zeilenumbruch ausgegeben werden. Das heißt, ein in ein CON:-Fenster ausgegebenes Wort wird nicht am rechten Ende des Windows abgeschnitten, sondern beginnt in der nächsten Zeile, wenn es nicht mehr in die letzte Zeile paßt. Wer mehr machen möchte als nur sehr einfache Textausgaben, sollte sich des Console-Devices bedienen.

15.4 Intuition-Screens und -Windows

Screens (virtuelle Bildschirme) und Windows (Fenster) sind rechteckig. Wenn wir einen Screen öffnen, deckt er die gesamte Breite des Bildschirms ab. Er braucht aber nicht unbedingt die gesamte Bildschirmhöhe abzudecken. Öffnen wir ein neues Window, ist seine Größe abhängig von den Werten in der Window-Datenstruktur. Wie Windows können auch die Screens durch Tiefen-Gadgets in den Vorder- oder Hintergrund gelegt werden.

Jeder Screen, den wir von einem eigenen Programm aus öffnen, kann jeden beliebigen Grafikmodus verwenden, also LoRes, HiRes, Interlace oder NichtInterlace, Single- oder Dual-Playfield-Modus und auch Hold & Modify. Für die hier genannten Grafikmodi gelten natürlich auch bei Programmierung unter Intuition die im Hardwareteil genannten Farbrestriktionen, also 32 Farben mit 5 Bitplanes bei LoRes, aber nur 16 Farben bei HiRes und so weiter. Das heißt, innerhalb eines Screens müssen die Farbkonventionen der Hardware beachtet werden. Da wir jedoch mehrere Screens gleichzeitig auf dem Bildschirm haben können, können wir auch mehrere verschiedene Grafikmodi gleichzeitig verwenden.

Jedes Programm kann mit Intuition ein oder mehrere »virtuelle Terminals« öffnen. Ein virtuelles Terminal ist einfach ein Window in einem Screen. Der Benutzer greift auf jedes virtuelle Terminal durch ein Window zu. Jedes Window »tut so«, als ob es das gesamte System für sich allein hätte. Unser Programm kann also bei der Programmierung unter Intuition ignorieren, daß verschiedene Ein-/Ausgaben in verschiedenen Fenstern gleichzeitig ablaufen können. Jedes unter Intuition geschriebene Programm kann so viele Windows öffnen, wie für seine Ein-/Ausgaben notwendig sind.

Auch die Änderungen des Benutzers am Window (Größenveränderungen, Verschieben) können vom Programm ignoriert werden, weil Intuition die sogenannten »Display Management Tasks« (Aufgaben des Bildschirm-Managements) automatisch erledigt. Will der Benutzer ein Window größer ziehen, braucht der Programmierer sich also nicht eine Abfrageroutine zu basteln, die ihm sagt, wie groß das neue Window ist und wie es neu aufzubauen ist. Programmierer anderer Systeme kennen die umständliche Arbeit, die hinter einer Fensterverwaltung ohne derartige Betriebssystem-Unterstützung liegt.

Ein Konzept, das Sie zur Programmierung unbedingt kennen müssen, ist das des »Beinhaltenden Display Elements«. Ein solches Display-Element ist ein Grafikelement, das ein weiteres Grafikelement beinhalten kann. Zum Beispiel ist ein Screen ein beinhaltendes Grafikelement; innerhalb eines Screens kann ein Window beinhaltet sein und im Screen positioniert werden. Ein Window ist ebenfalls ein beinhaltendes Grafikelement: es kann einen Requester beinhalten, der im Window positioniert werden muß.

Wenn ein Element in das Display hinzugefügt wird, wird es immer innerhalb eines anderen Elements plaziert. Das zeigt sich nicht nur in der Grafik, sondern auch in den Mechanismen der Arbeit mit den Datenstrukturen, die die Display-Elemente definieren. Im Normalfall wird zum Beispiel die Koordinatenangabe eines Elements auf die linke obere Ecke (0,0) des beinhaltenden Elements bezogen.

Die »Offset-Position« (das heißt die Koordinaten des Elements in bezug auf die linke obere Ecke) zeigt sich normalerweise in der Struktur eines Display-Elements als die »TopEdge«- und »LeftEdge«-Parameter. Diese Parameter sind beispielsweise in den Strukturen *Screen, NewScreen, Window, NewWindow, Menu,*

MenuItem, Requester, Gadget, Border und *Image* enthalten (Namen der Datenstrukturen, die von Intuition verwendet werden). Das heißt, immer wenn wir ein neues Display-Element öffnen, müssen wir das beinhaltende Element im Hinterkopf behalten, um die Position des neuen Grafikelements richtig angeben zu können.

Die x-Position eines Screens ist die linke Seite des Bildschirms, die y-Position des Screens kann zwischen dem Bildschirmanfang und dem Bildschirmende liegen. Windows können an beliebiger Position innerhalb des Screens liegen und dürfen sich auch beliebig überlappen. Screens können vertikal, aber nicht horizontal bewegt werden. Windows können innerhalb eines Screens ohne Beschränkung in jede Richtung bewegt werden. Die Bewegung von Screens oder auch Windows kann unabhängig davon vorgenommen werden, ob sie gerade von anderen Screens beziehungsweise Windows überlappt werden.

Es kann aber immer nur ein Window zu einer gegebenen Zeit aktiv sein (Wie Sie sicher auf der Workbench schon bemerkt haben, ist eine gleichzeitige Eingabe in zwei Windows nicht möglich). Das *aktive* Window ist dasjenige, das vom gerade aktiven Task für Ein- und Ausgaben benutzt wird. In anderen Windows können weitere Tasks ablaufen, solange keine Eingabe vom Benutzer erwartet wird. Eine Eingabe des Benutzers in das aktive Window kann allerdings vom Programm aus auch an andere Windows zur Verarbeitung weitergeleitet werden. Das heißt, Eingaben bewirken in mehreren Windows etwas, obwohl nur ein einziges wirklich aktiv ist.

Alle Windows sind Screens zugeordnet. Alle Windows, die einem bestimmten Screen zugeordnet sind, nehmen dessen Display-Charakteristiken an, die für diesen Screen festgelegt wurden, was natürlich die Farben und den Grafikmodus einschließt.

Der Mauspfeil, auch Mauszeiger oder »Selection Pointer« genannt, kann innerhalb eines Screens aber auch über Screengrenzen hinweg bewegt werden. Durch Anklicken eines Windows mit dem Zeiger kann der Benutzer bestimmen, welches der sichtbaren Windows das aktive ist. Man ist für den Mauszeiger nicht auf eine Form angewiesen, sondern kann auch eigene Formen verwenden. Hierfür stehen die Funktionen *SetPointer* und *ClearPointer* zur Verfügung. Setzt man keinen eigenen Zeiger fest, wird der Standard-Intuition-

Zeiger benutzt. Dieser Zeiger sieht aus wie der rote Pfeil, der im Normalfall auch von der Workbench benutzt wird.

Arten von Screens

Es gibt zwei Arten von Screens im Intuition-System: Standard- und Custom-Screens. Im Augenblick ist der Workbench-Screen der einzige Standardscreen (das Intuition-System wurde aber so gebaut, daß Erweiterungen für zukünftige Amiga-Modelle möglich sind. Soll beispielsweise ein Gerät mit höherer Auflösung gebaut werden, wird für dessen Auflösung ein weiterer Standardscreen entwickelt. Durch Anwählen des alten Standardscreens ist jedoch Software für die alte Workbench noch kompatibel). Custom Screens sind selbstdefinierte Screens – also alle anderen Screens außer der Workbench.

Jede andere Intuition-Grafik-Komponente (Windows, Gadgets, Requesters und so weiter) ist durch die Werte des Screens, dem sie zugeordnet ist, definiert. Wenn ein Programm Windows mit mehreren verschiedenen Charakteristiken (Grafikmodi und so weiter) benötigt, müssen dafür verschiedene Screens geöffnet werden und die Windows innerhalb dieser Screens geöffnet werden.

16

Die Grafiklibrary

Unter den Grafiklibraries des Amiga verstehen wir eigentlich sowohl die Grafikprimitive, wie *Zeichnen, Bildschirm öffnen* und so weiter, aber auch die Animations- und Textfunktionen. Uns sollen im Rahmen dieses Kapitels jedoch vorerst nur die einfachen Grafikfunktionen interessieren.

Diese Funktionen lassen sich in zwei große Gruppen unterteilen: Die Display- und die Zeichenfunktionen. Die Displayfunktionen dienen in erster Linie dazu, die Ausgabe vorzubereiten. Dazu gehört es, Bitmaps und Bitplanes zu initialisieren, den Speicher dafür zu verwalten und schließlich die Anzeige des Bildes zu veranlassen. Die Zeichenfunktionen dienen zur Darstellung der eigentlichen Elemente einer Computergrafik. Sie setzen Punkte, ziehen Linien, füllen Flächen und so weiter.

16.1 Die Anzeige- oder Display-Funktionen

Die Displayfunktionen reservieren als allererstes Speicher für die Grafik aus den gewünschten Bitplanes und Bitmaps. Diese RAM-Bereiche werden mit Werten initialisiert, die von den aufgerufenen Betriebssystemroutinen vorgegeben wurden. In den so reservierten Bitmaps ist allerdings noch keine echte Bildinformation enthalten; es handelt sich zuerst einfach nur um gelöschte Bereiche des RAMs. Dies ergibt einen leeren Bildschirm. Wir rufen dazu nur die Funktionen zum Öffnen des Bildschirms auf. Die Grafiklibrary ruft dann von sich aus die Exec-Routine zum Reservieren des Speichers auf und löscht anschließend diesen Bereich, um ein sauberes Bild zu geben.

Beim Öffnen einer Grafik müssen wir sehr genau darauf achten, wie der Bildaufbau erfolgen soll. Wenn Sie den Hardwareteil aufmerksam durchgelesen haben, wissen Sie ja, daß der Copper jede 1/60-Sekunde ein Bild aufbauen kann. Beim Aufbauen unseres Displays setzen wir das Gesamtbild in einen sogenannten »View« (ein View ist ein rechteckiger Teil des Bildes, der die volle Breite des Bildschirms hat, aber nicht notwendigerweise die volle Höhe haben muß). Jeder View kann aus unterschiedlichen »Viewports« bestehen. Ein Viewport ist ebenfalls ein rechteckiger Teil des Bildes, kann aber auch in der Breite kleiner sein als der Bildschirm. Das Ergebnis davon ist also ein aus mehreren Viewports bestehender View. Dieser View wird definiert durch seine Copper-Instruktionen. Das heißt, wenn wir nur einen einzelnen View als komplettes Bild darstellen wollen, brauchen wir seine Copper-Befehlsliste nur jede sechzigstel Sekunde an die Display-Hardware übergeben. Wollen wir das Display in verschiedene Teile aufspalten (ähnlich den Intuition-Screens), müssen wir mehrere Views entwickeln und deren Definition durch die entsprechende Copperlist-Befehle an die Hardware übergeben.

Die Displayfunktionen fallen in acht Kategorien: Die Bitmap-Funktionen, die Superbitmap-Funktionen, die RastPort-Struktur-Initialisierungs-Funktion (die RastPort-Struktur ist die wichtigste Datenstruktur des gesamten Grafiksystems. Sie enthält alle wichtigen Informationen zum Bildaufbau), die View- und die Viewport-Funktionen, die Region-Definition-Routinen, die Layerfunktionen (siehe unten) und die Hardwarekontroll-Funktionen. Die Hardwarekontroll-Funktionen zerfallen wiederum in eine Anzahl von Untergruppen: Die Copper-Kontroll-Funktionen, die Blitter-Kontroll-Funktionen, und die Funktionen zur Rasterstrahlpositions-Überprüfung und -Kontrolle.

16.2 Die Zeichenfunktionen

Die eigentlichen Bilder, die wir auf dem Bildschirm sehen, also Kreise, Linien und so weiter, werden durch die Zeichenfunktionen in die Bitmaps gesetzt. Die Grafik-Zeichen-Routinen (beispielsweise die Funktion *WritePixel* zum Setzen einzelner Punkte) rufen wir immer mit einem Zeiger auf die RastPort-Struktur der gewünschten Bitmap auf. Für jede Bitmap haben wir also eine RastPort-Struktur definiert. In dieser stehen Informationen über den BitMap-Speicher,

die Speicherstellen, in denen Füllmuster für Füllbefehle liegen, Zeiger auf Bobs und Sprites für die Einbeziehung in das Bild, Farbinformationen und so weiter. Dem Programmierer werden zum Zeichnen Routinen zur Verfügung gestellt, die von der tiefsten Ebene (zum Beispiel *WritePixel*) bis zu höherwertigen Grafikfunktionen (*Floodfill, Polygone* und so weiter) reichen.

Die Zeichenfunktionen fallen in sieben Kategorien: Die Pixelfarbe-Lese- und -Schreibfunktionen, die Zeichenstiftfarbe-Kontrollfunktionen, die RGB-Funktionen, die Zeichenmodus-Funktionen, die Areafillfunktionen, die Colormap-Funktionen und die Funktionen zur Behandlung ganzer Grafikbereiche.

Neben den einzelnen Zeichenbefehlen gibt es auch noch Makros in der Grafikbibliothek. Die Makros fassen einzelne Befehle zusammen und erledigen größere Aufgaben. Darunter fallen zum Beispiel die Coppermakros (CINIT, CMOVE, CWAIT), die es erlauben, die Copperlisten der Hardware direkt zu beeinflussen. Die Set-Makros (*SetOPen, SetWrMsk* und so weiter) erlauben direkten Zugriff auf die Informationen der RastPort-Struktur einer Bitmap. Die Makros werden wie alle anderen Funktionen der Grafikbibliothek aufgerufen und rufen ihrerseits weitere Routinen des Systems in vorausbestimmter Reihenfolge auf. Das ist in etwa vergleichbar mit den Execute-Files des CLI.

16.3 Koordinatensysteme des Betriebssystems

Die Funktionen der Grafiklibrary arbeiten nicht alle mit demselben Koordinatensystem. Wir müssen bei jeder Funktion zunächst einmal nachsehen, mit welchem wir arbeiten wollen. Das ist keine reine Willkür der Systemprogrammierer, sondern ein Vorteil für den Programmierer. Die verschiedenen Koordinatensysteme machen verschiedene Aufgaben möglichst einfach oder schnell.

Das »Rasterkoordinatensystem« ist das größte der Intuition-Koordinatensysteme. Die (x,y)-Werte reichen hier bis zur Position (1024,1024). Die meisten Zeichenfunktionen, darunter *Draw, Move, Polydraw* und die Area-Funktionen, verwenden dieses Koordinatensystem. Damit ist auch gewährleistet, daß in großen Superbitmaps (größer als der Bildschirm) Grafiken ohne viel Rechnerei erstellt werden können.

Das View-Koordinatensystem ist das nächstgrößere. Es wird auch »Display-Frame-System« genannt. Das Größenmaximum des View-Systems hängt vom Grafikmodus ab (der mit der *SetDrMod*-Funktion eingestellt wurde). In einem Standard-LoRes-NonInterlace-Bild liegt die obere linke Ecke dieses Koordinatensystems bei (0,0) und die untere rechte Ecke bei (320,200). In Custom-Screens kann man aber auch (im HiRes-Interlace-Modus) Koordinaten bis (704,464) erreichen. Auf den meisten Monitoren sieht das aber – wegen des bereits erwähnten Overscan-Effekts – nicht sehr gut aus. Nichtsdestotrotz können wir auch die 704x464 Punkte nutzen. Das Softwarepaket »Caligari« von Octree-Software aus den USA zeigt das deutlich und recht eindrucksvoll.

Das kleinste Koordinatensystem schließlich ist das Viewport-System. Die maximalen Koordinaten dieses Systems hängen von der Größe des jeweils größten Viewports ab. Damit wissen Sie nun auch endlich, wozu die Viewports gut sind: Kleine Rechtecke in der Hintergrundgrafik können als Koordinatensysteme verwendet werden, ohne daß wir dafür die speicherplatzaufwendigen Windows verwenden müssen. Die Grafik der Viewports fügt sich fugenlos in die Hintergrundgrafik ein, kann aber eben fast wie ein Window mit seinem eigenen Koordinatensystem behandelt werden. Nur brauchen wir hierzu keine Intuition-Bibliothek aktivieren.

17

Die Animationsfunktionen

Die Animationsfunktionen des Amiga sind ein sehr wirkungsvolles Mittel für die Handhabung bewegbarer Objekte. Allerdings funktioniert nicht alles immer so, wie es sollte – beziehungsweise in den entsprechenden Handbüchern beschrieben wird. Deshalb findet man auch in der neuesten Ausgabe des Amiga ROM Kernel Manuals (September 86) ein Beispiellisting in C, das gezielt die bislang nicht funktionierenden Teile der VSprite-Routinen umgeht. Na ja, immerhin wissen wir jetzt, daß man die VSprites doch benutzen kann. Zudem sind die Animationsroutinen der Animationsbibliotheken nicht unbedingt die allerschnellsten, denn es wird jede kleinste Möglichkeit und auch Multitasking berücksichtigt – was natürlich enorm am Zahn der Zeit nagt. So schrieb beispielsweise der Autor des Spiels »Space Battle« seine eigenen Routinen, um eine annehmbare Geschwindigkeit der BOBs zu erreichen.

Nun wollen Sie natürlich wissen, was die eben besprochenen »Vsprites« sind; BOBs sollten Ihnen nach Lektüre des BASIC- oder Hardwareteils inzwischen ein Begriff sein. Die Animationsbibliotheken umfassen neben diesen beiden Komponenten aber noch einige andere Animationsobjekte, die im folgenden kurz beschrieben werden.

17.1 Betriebssystemunterstützte Animationsobjekte

Es gibt fünf Arten von betriebssystemunterstützten Animationsobjekten. Da sind einmal die Hardware Sprites, die sogenannten virtuellen Sprites, die altbekannten BOBs, sogenannte »AnimComps« und schließlich die »AnimObjects«.

17.1.1 Hardware Sprites

Hardware Sprites sind die ganz normalen Sprites, wie ich sie auch im Hardwareteil des Buches beschrieben haben. Sie sind 16 Pixels breit und können so lang (hoch) wie der gesamte Bildschirm sein. Wie Sprites genau aufgebaut sind und wie sie funktionieren, lesen Sie bitte im Hardwareteil nach.

Jedes Hardwaresprite wird durch zwei Datenstrukturen »SimpleSprite« und »SpriteImage« definiert. Die Hardwaresprite-Funktionen zum Steuern der Sprites heißen *GetSprite*, *ChangeSprite*, *FreeSprite* und *MoveSprite* und sind relativ einfach zu handhaben.

Die Hardware-Sprites können allerdings nicht in Animation höherer Systemlevel wie die AnimComps und AnimObjects (siehe unten) integriert werden.

17.1.2 Virtuelle Sprites

Virtuelle Sprites sind im Grunde auch Hardwaresprites. Sie werden allerdings dadurch erzeugt, daß ein Sprite-DMA-Kanal mehrmals verwendet wird. So können wir mit ein- und demselben Spritekanal mehrere Sprites darstellen – mehrere Sprites desselben DMA-Kanals dürfen allerdings nie dieselbe y-Position belegen, denn beim Durchlauf des Rasterstrahls durch eine Zeile kann von einem DMA-Kanal auch immer nur ein Sprite dargestellt werden. Wie diese virtuellen Sprites von der Hardware verwaltet werden, finden Sie ebenfalls im Hardwareteil dieses Buches (im Kapitel über Sprites) erläutert. Die Verwaltung der VSprites brauchen wir als Programmierer allerdings nicht übernehmen. Die Systemsoftware weist die entsprechenden Sprites automatisch den jeweils freien

DMA-Kanälen zu. Die hierfür nötigen Routinen heißen zum Beispiel *AddVSprite* oder *RemVSprite*. Jedes VSprite ist durch eine VSprite-Datenstruktur festgelegt.

Der Vorteil der VSprites gegenüber den eigentlichen Hardwaresprites ist die große Vielfalt der Objekte und Farben, die man auf einem Bild darstellen kann. Außerdem können wir Kollisionen zwischen den VSprites und anderen Objekten des Animationssystems feststellen. Die Systemsoftware zu den Hardwaresprites bietet diese Möglichkeit nicht. VSprites können allerdings leider auch nicht in die AnimComps und AnimObjects mit einbezogen werden.

17.1.3 BOBs – Blitter Objects

BOBs können aus Softwaresicht eigentlich fast wie VSprites behandelt werden. Der Unterschied ist, daß die BOBs nicht von DMA-Kanälen »über« das eigentliche Bild gezeichnet werden, sondern vom Blitter in die Bitplane des Hintergrundbildes (Playfields) einkopiert werden. BOBs sind nicht so schnell wie VSprites, können dafür aber breiter als 16 Pixels und auch beliebig hoch werden. Die Farbenvielfalt ist dabei außerdem nicht so beschränkt. So viele Farben, wie der Hintergrund hat, kann auch ein BOB haben – bis zu 32 verschiedene Farben also.

Die BOB-Routinen entsprechen in der Arbeitsweise und auch der Namensgebung in etwa denen der VSprites. *AddBob* und *RemBob* heißen die wichtigsten BOB-Routinen. BOBs können in Gruppen zu AnimComps zusammengefaßt werden. Und die AnimComps wiederum können in AnimObjects gruppiert werden.

17.1.4 AnimComps

AnimComps sind mehrere BOBs, die in einer Gruppe zusammengefaßt werden. Ein Segelschiff besteht zum Beispiel aus Bug, Ruder, Mast und Segeln. Jedes Teil davon kann ein einzelnes BOB sein, doch zusammen ergeben die BOBs die Animationskomponente (»AnimComp«) *Schiff*.

Jede einzelne AnimComp ist durch eine sogenannte AnimComp-Struktur definiert, die genau sagt, welche Objekte miteinander verbunden sind. Diese AnimComp-Struktur verfügt auch über einen Satz

von Timing-Parametern, die die jeweilige »Lebenszeit« (Dauer des Erscheinens) der Animationskomponente kontrollieren, wenn sie in einem AnimObject verwendet werden. Die AnimComps selbst enthalten in ihrer Datenstruktur keine weiteren animationsspezifischen Komponenten. Diese werden alle schon in den BOB-Strukturen definiert.

17.1.5 AnimObjects

AnimObjects (Animationsobjekte) sind die komplexesten Animationselemente der Animationsbibliotheken des Amiga. Dabei handelt es sich um Gruppen von AnimComps, die miteinander »verbunden« sind. Der Programmierer braucht nur noch ein AnimObject zur Bewegung veranlassen, was automatisch alle AnimComps mitbewegt. So kann er also durch einen einzigen Befehl eine ganze Menge bewirken.

Für die Bewegung von AnimObjects gibt es zwei grundsätzlich verschiedene Methoden: die sogenannte »sequenced drawing animation« und die »motion control animation«.

Bei der Methode 1, der »sequenced drawing animation« (wörtlich übersetzt »sequentielles Zeichnen«, was den Kern der Sache recht gut trifft), muß der Programmierer eine Folge (Sequenz) von Positionen für AnimObjects definieren. Diese Objekte werden dann in der Reihenfolge auf den Bildschirm gebracht, wie sie in der AnimOb-Struktur stehen. Die oben bereits angesprochenen Timing-Parameter kontrollieren dann, wie lange das gesamte AnimObject auf der entsprechenden Position des Bildschirms bleibt. Mit dieser Methode können wir beispielsweise Formationen für die Bewegung von Ufos in einem Spiel wie »Galaxians« festlegen.

Methode 2, die »motion control animation«, verlangt vom Programmierer, Startposition, Geschwindigkeit und Beschleunigung von AnimObjects festzulegen. Wie der Name »motion control« schon sagt, kontrollieren wir hier nicht nur einige Positionen, sondern wirklich eine Bewegung. Im Gegensatz zur sequentiellen Methode wird hier der Bewegungsablauf nicht auf festen »Schienen« vorgegeben, sondern wird mathematisch ermittelt. Die Motion-Control-Methode wird übrigens auch von den OBJECT-Befehlen des AmigaBASIC verwendet. Vielleicht wird Ihnen nun auch klar, warum die BASIC-

Routinen so langsam sind. Jedes einzelne BOB wird hierzu nämlich als AnimComp deklariert. Will ich eine ganze Gruppe von Objekten gleichzeitig bewegen (beispielsweise eine ganze Gruppe von »Space Invaders«), würde ich als Betriebssystemprogrammierer alle zu einem AnimObject zusammenfassen, das wiederum aus zwei AnimComps besteht. In jeder AnimComp ist eine der zwei Bewegungsphasen eines Space Invaders zusammengefaßt. Für die Bewegung des ganzen Invader-Bataillons müßte ich also nur einen einzelnen Aufruf für das ganze AnimObject verwenden. BASIC muß für jeden einzelnen der Invaders die Animate-Funktion aufrufen. Deshalb scheiterte übrigens mein Versuch der Programmierung eines Space-Invader-Spieles in BASIC kläglich.

17.2 Die Verwaltung der Animation durch das System

Neben den Funktionen für Hardware-Sprites, VSprites, BOBs, Animation-Components und -Objects sind noch einige weitere Funktionen vorhanden, die für bewegte Objekte sehr wichtig sind. Außer dem Starten der Bewegung mit der Animate-Routine müssen wir uns nämlich noch um Kollisionen der Objekte und das Management der Objekte durch die Software kümmern. Das allerwichtigste dabei ist natürlich, daß uns Speicher zur Verfügung stehen. Dazu gibt es die Funktionen *GetGBuffers* und *FreeGBuffers*, die nur so viel Speicherplatz reservieren, wie für die aktuelle Animation gerade notwendig ist.

Die Kollisionen werden durch die sogenannten »Gel-to-Gel-Collision-Management«-Routinen verwaltet. (Was ein GEL ist, erfahren Sie in den folgenden Ausführungen über das Animationsobjekte-Management.) Kollisionsroutinen sind beispielsweise *SetCollision* und *DoCollision*. Schließlich gibt es dann auch noch Listenmanagement-Funktionen für die Verwaltung der einzelnen Objekte als Gesamtheit. *InitGels*, *SortGList* und *DrawGList* heißen die entsprechenden Routinen.

17.2.1 Objektlisten-Management

Damit das System die Grafikobjekte möglichst schnell und effizient darstellen und bewegen kann, müssen Listen der Elemente angelegt und schließlich auch sortiert werden. Schließlich soll das am Bildschirm oberste Objekt logischerweise zuerst dargestellt werden, dann das darunterliegende und so weiter, bis wir beim untersten Objekt des Bildschirmes sind. Der Rasterstrahl kann ja schließlich nicht dauernd auf- und abhüpfen. Und das Herausfinden eines Objektes, das eine höhere y-Position hat als alle anderen, aus einer ungeordneten Menge Objekte, ist einfach zu langsam.

Die wichtigste Liste ist die sogenannte GELS- oder »Graphics Elements«-Liste. In ihr sind alle Grafikelemente (VSprites und BOBs) zusammengefaßt. Die Funktion *SortGList* ordnet die Elemente in dieser Liste in aufsteigender y,x-Reihenfolge. Die Funktion *DrawGList* erzeugt dann die entsprechenden Copper-Befehle, um das Display mitsamt den Grafikelementen von oben nach unten zeilenweise aufzubauen. GELS sind also alle Grafikelemente!

Die Reihenfolge des Vorgehens ist also: Definieren der Objekte, Eröffnen einer GELs-Liste mit der Funktion *InitGels* und Anfügen der entsprechenden Objekte mit den *Add*-Funktionen (*AddBob, AddVSprite*). Die Parameter zum Einbinden der Objekte in die richtige Stelle der Liste sind in den entsprechenden Datenstrukturen für Bobs und VSprites enthalten.

Weitere Listen sind die Bob-Listen, die die Prioritäten der einzelnen Bobs untereinander definieren (welches Bob liegt »über« welchen anderen Bobs). Außerdem existieren separate Listen für die einzelnen AnimComps und AnimObjects. Darin werden unter anderem auch Reihenfolgen der Darstellung in einer »sequenzed drawing«-Animation festgehalten.

Die Animationsroutinen des Amiga sind die wohl kompliziertesten, die es im ganzen System gibt. Ich muß zugeben, daß ich selbst nicht immer so recht damit umgehen kann. Anfängern sei also empfohlen, zuerst einmal mit Intuition zu experimentieren, dann auf die Grafiklibrary zuzugreifen, und sich der Animation erst dann zu widmen, wenn alles andere auch so funktioniert, wie man es will.

18 Sonstige Grafikfunktionen

18.1 Die Grafik-Text-Funktionen

Beim Amiga haben wir es im Vergleich mit einigen anderen Computern nicht mit einem System zu tun, das die Buchstaben per Hardware (mit einem Character-Generator-Chip) ins Bild bringt. Stattdessen werden die Buchstaben per Software in den Grafikschirm einkopiert. Text ist auf dem Amiga also nichts anderes als ein Bestandteil der Hintergrundgrafik (des Playfields).

Die Textfunktionen des Amiga-Systems im einzelnen zerfallen in zwei Gruppen: die Routinen der DiskFont-Library und die Textfunktionen der Grafik-Library. Die Grafik-Library enthält Textmanagement-Funktionen (*ClearScreen, Text, TextLength, ClearEOL*) zum Plazieren von Texten in die Bitmaps und Font-Management-Funktionen für RAM-residente Zeichensätze (*AddFont, AskFont, OpenFont, SetSoftstyle* und so weiter) für das Hinzufügen, Wegnehmen und Verwalten von Fonts im RAM. Die Font-Management-Funktionen für die Verwaltung von Disketten-residenten Zeichensätzen (*AvailFonts, OpenDiskFont*) befinden sich in der seperaten Library »DiskFont«.

Das Verwalten von Text mit Hilfe des Betriebssystems ist allerdings nicht so einfach wie in einer höheren Programmiersprache, also zum Beispiel wie in BASIC und Pascal. Hier müssen wir uns ebenfalls wieder recht intensiv um Listen und Datenstrukturen kümmern. Die TextFont-Struktur enthält beispielsweise alle wichtigen Daten zur Definition der Fontparameter und Zeiger auf die Grafikdaten der einzelnen Buchstaben. Und um die Speicherplatzreservierung für die

Fonts müssen wir uns ebenfalls kümmern. Alle Fonts, die mit der *AddFont*-Routine in die RAM-Fonts eingebunden wurden, müssen wieder in einer Liste festgehalten werden. Dabei kann die Verwendung bestimmter Fonts auch in manchen Tasks verboten werden, so daß jeder Task nur die ihm zustehenden Zeichensätze bekommt. Für jeden Task, der Texte ausgibt, setzen wir hierzu einfach einen Zeiger auf eine andere Font-Liste. In Datenstrukturen wie *TextAttr* (Text-Attribute) können wir auch die verschiedenen Arten eines Fonts bestimmen, also zum Beispiel Unterstreichungen, Fettdruck oder kursive Schrift.

Beim Ausgeben von Text können wir die Position des Textes auf Pixels genau angeben (mit der Routine *Text*). Die Farben eines so ausgegebenen Textes werden durch die *Pen*-Befehle der Grafiklibrary definiert (Text erscheint immer in der Vordergrundfarbe).

18.2 Die Layer-Funktionen

Die Layers-Library ermöglicht es erst, daß die Verwaltung von Fenstern in Intuition so wunderbar funktioniert. Die von Intuition unterstützten Refresh-Techniken für Windows werden von der Layers-Library zur Verfügung gestellt. Intuition greift also auf die Layers-Library zu. Wenn wir die Intuition-Screens zum Beispiel mit den Depth-Gadgets vor- oder rückblenden, ruft Intuition über seine eigenen Routinen *ScreenToFront* und *ScreenToBack* auf, die dazu passenden Funktionen der Layers-Bibliothek.

Was sind nun eigentlich diese Layers? Kurz gesagt, es sind übereinander liegende (Schichten von) Grafiken. Die Layers-Library kümmert sich vor allem um die Probleme, die auftreten, wenn sich verschiedene Grafiken überlappen. Dabei ist allerdings nicht der Grafikschirm in diese Überlappungsteile (Layers) aufgeteilt, sondern nur der Speicher. Die Layers-Library kopiert nur die verschiedenen Layers des Speichers in den Display-Screen.

Tatsächlich können wir die Layers, die der Intuition-Benutzeroberfläche zugrunde liegen, selbst für eigene Zwecke verwenden und mißbrauchen. Die Funktion *WhichLayer* sagt uns beispielsweise, aus welchem Layer im Speicher eigentlich die Grafik auf einer bestimmten Position des Bildschirms stammt. So wissen wir, in welcher Layer-Bitmap wir etwas ändern müssen, wenn sich die Maus an

dieser Stelle befindet und die darunterliegende Grafik geändert werden soll, das jeweilige Window aber nicht durch Mausklick aktiviert wurde (denn nur dann weiß Intuition, um welches Fenster es sich handelt).

Die Layers-Bibliothek stellt uns auch einen Befehl zum Scrollen von Superbitmaps zur Verfügung (*ScrollLayer*). Dieser Scroll-Befehl kopiert sehr schnell die ausgewählten Bereiche einer Superbitmap in ein Window, wie auch beim Lines-Demo der Workbench 1.2 zu sehen ist. Für die Kontrolle einer großen Grafik ist diese Technik sehr einfach und zudem recht schnell.

Die Layers-Library erlaubt es außerdem, den Zugriff von einzelnen Tasks auf Layers zu verbieten oder ausschließlich bestimmten Tasks zu erlauben. Damit ist sichergestellt, daß nicht ein Programm zufällig in denselben Bildspeicher schreiben kann wie ein anderes. Wenn sich zwei Tasks ein Window teilen, ist es auch oft wichtig, die Reihenfolge der Grafikoperationen zu regeln. Auch das können wir mit den entsprechenden Layer-Routinen erreichen (zum Beispiel Routinen *LockLayer* und *UnlockLayer*).

Auch die Layer-Library wird durch Datenstrukturen gesteuert. Die Struktur *Layer* selbst definiert die Charakteristiken einzelner Layers (Simple-Refresh, Smart-Refresh, Superbitmap, Backdrop) und die Verbindungen eines Layers mit anderen. Die *LayerInfo*-Struktur liefert die wichtigen Daten für den Mechanismus zur Kontrolle von Zeichenoperationen in Layers. Damit wird auch das Task-Management für die Grafikbereiche geregelt. Die *ClipRect*-Struktur hat das Ziel, eine Liste von sogenannten »Clipping-Rechtecken« zu verwalten. Dies ermöglicht es, Grafiken, die über die Grenzen eines Layers hinausgehen, abschneiden zu können. Zudem setzt die ClipRect-Struktur auch die Teile eines Layers fest, die beim nächsten Bildaufbau neu gesetzt (aufgefrischt) werden müssen.

18.3 Sonstige Betriebssystemroutinen zur Grafik

Neben den eigentlichen Grafikfunktionen, die ich in den letzten Kapiteln hoffentlich gut genug erklärt habe, gibt es noch einige weitere Funktionen, die im Zusammenhang mit Grafik interessant erscheinen. Die Library »Icon«, die die Workbench-Funktionen bein-

haltet, kümmert sich beispielsweise um die schönen Icons (Pictogramme), die Sie als Benutzer auf der Workbench herumschieben können. Sie sind ein nicht zu vergessender Teil des optischen Eindrucks, den ein Programm beim Benutzer hinterläßt. Leider gibt es keine Routinen, mit denen man die Workbench-Icons bewegen kann – alles, was man tun kann, ist es, sie von Diskette zu holen und dann auf dem Schirm darzustellen. Die Arbeit mit dem Icon-Editor ist in Ihrem Handbuch ausführlich beschrieben – experimentieren Sie ruhig ein bißchen damit.

Der direkte Zugriff auf die Grafikhardware durch die Resources könnte ebenfalls für Sie von Interesse sein. Und schließlich sind auch die Fast Floating-Point-Routinen für den Grafikprogrammierer von Interesse, die eine hervorragende Hilfe bei der Berechnung komplexer Grafikelemente (insbesondere 3D-Grafiken) darstellen.

Buchteil 5

In diesem Teil des Buches sollen alle Anwender und solche die es werden wollen Gelegenheit erhalten, einen umfassenden Überblick über die derzeit erhältlichen Soft- und Hardwareprodukte zur Amiga-Grafik zu bekommen.

Die angesprochenen Softwareprodukte sind keine Spielprogramme, sondern professionelle Programme und können durchaus auch für Aufgaben in Videostudios, Architekturbüros und anderen anspruchsvollen Anwendungen verwendet werden.



19 Professionelle Grafiksoftware

Für den Amiga existiert inzwischen eine Unmenge an Software, die zumeist auch die Grafikfähigkeiten dieses Computers ausnutzen. Doch nur wenige Programme haben einen hohen »Nutzungsgrad« dieser Fähigkeiten. Wer verwendet schon die 4096 Farben des HAM-Modus in einer schnellen Animation? Eigentlich niemand, doch das soll sich Gerüchten zufolge bald ändern: Die Firma Aegis Software arbeitet an einem Animationspaket auf Basis des Hold & Modify-Modus und extrem kompakter Bildspeicherung. Auch ein anderer Hersteller hat bereits ein Malprogramm für den 4096-Farben-HAM-Modus angekündigt. Doch gehen wir weg von den Gerüchten und sehen uns die bereits verfügbare Grafiksoftware an.

19.1 Zeichen- und Malprogramme

Unter Malprogrammen verstehen wir Software, mit der frei in einer Bitmap-Grafik nach dem Frame-Buffer-Prinzip gezeichnet werden kann. Was man auf dem Bildschirm gezeichnet hat, wird also auch genau so im Grafikspeicher abgelegt. Im Gegensatz dazu existieren die sogenannten »objektorientierten« Zeichenprogramme, die nicht das Bitmap-Format, sondern die geometrischen Grundformen (Kreise, Rechtecke, Linien und so weiter) abspeichern. Näheres dazu finden Sie unter der Überschrift »CAD«-Software.

Das erste Malprogramm, das auf dem Amiga erhältlich war, setzte bereits einen neuen Standard für die Leistungsfähigkeit eines Grafikprogrammes für Personal Computer. Vorgestellt wurde es bereits vor der Markteinführung des Amiga in Europa, auf einer Soft-

ware-Entwickler-Konferenz in Großbritannien. Wer den Amiga-Markt kennt, weiß, wovon ich spreche: DeluxePaint von Electronic Arts. Kurz darauf konnte man bei Commodore schon die ersten Exemplare eines Grafikprogrammes ganz anderer Leistungsklasse finden: Graphicraft war ein einfaches Zeichenprogramm für den Heimgebrauch, das dafür aber extrem preiswert ist. Doch andere Softwarefirmen schlafen auch nicht, und schon bald stellte Aegis Software das Programm »Images« vor, das kaum einen Vergleich mit DeluxePaint zu scheuen braucht. Schon trafen aber schlechte Nachrichten für Aegis ein: Electronic Arts arbeitete an einer neuen Version des DeluxePaint (die inzwischen auch erschienen ist). Seit Anfang des Jahres gibt es nun auch endlich das erste Buch zur Arbeit mit DeluxePaint und den anderen Programmen der Deluxe-Serie auf dem Markt (Markus Breuer, »Deluxe-Grafik mit dem Amiga«, bei Markt&Technik).

19.1.1 Deluxe Paint II

DeluxePaint II, das wohl leistungsfähigste Grafikprogramm für Mikrocomputer, ist so professionell, daß es bereits in Videostudios für die Bearbeitung und Manipulation von Text- und Grafikeinblendungen in das Videobild verwendet wird.

Wird das Programm geladen, kann man als erstes die Grafikauflösung und Anzahl der Farben auswählen. Alle Auflösungen des Amiga werden unterstützt. Befindet man sich dann erst einmal im Programm, ist es sogar möglich, die Seitengröße zu verändern. Das bedeutet: ein Bild kann größer sein, als der Bildschirm. Dabei kann man den jeweils sichtbaren (und bearbeitbaren) Teil des Bildes mit den Cursortasten verschieben. Im *Page*-Menü gibt es drei Standard-Einstellungen für die Seitengröße. Die eine nennt sich *Full Page* und ergibt in etwa eine DIN A4-Seite. Die zweite heißt *Full Video* und die dritte schließlich beschränkt sich auf die jeweils eingestellte Bildauflösung. Die Möglichkeit *Full Video* nutzt die volle Video-Auflösung, die der Amiga zustandebringt; im HiRes-Modus und mit Interlace sind das 672 x 444 Punkte. Das geht auch auf den amerikanischen Amigas. DeluxePaint II erkennt ansonsten zwar automatisch auch die europäische PAL-Norm, erreicht aber trotzdem nicht die damit möglichen 704 x 512 Punkte. Die ganze Seite läßt sich übrigens mit einem einfachen Befehl auf einmal – aber in

entsprechend verkleinerter Form – auf dem Bildschirm anzeigen, egal wie groß die Seite jeweils ist. Die mögliche Größe einer Seite hängt von der Anzahl der Farben, der Auflösung und natürlich dem Vorhandensein oder Nichtvorhandensein einer Speichererweiterung ab. DeluxePaint II läuft hervorragend auf einem Standard-Amiga mit 512 KBytes RAM, er kennt aber auch Speichererweiterungen. Maximal nutzt das Programm 1 000 KBytes (1 MByte). Mehr als das wird nicht benötigt, um eine 1024 x 1024 Punkte große HiRes-Seite in 16 Farben darzustellen.

Deluxe Paint bietet ansonsten alle Standard-Funktionen eines guten Malprogrammes, also Punkte, Linien, Rechtecke, Kreise, Ellipsen und Polygone – natürlich auch auf Wunsch ausgefüllt mit einer bestimmten Farbe oder einem Muster. Das Muster wird einfach aus einem sogenannten »Brush«, einem Pinsel geholt. Der Clou des Programmes ist es nämlich, jeden beliebigen Teil eines Bildes als Pinsel verwenden zu können. Mit diesem Pinsel kann gezeichnet werden, er kann in Füllmuster umgewandelt werden, und man kann ihn auch auf andere Weise manipulieren.

Zudem gibt es Funktionen wie Airbrush (Sprühdose) oder das Zeichnen von gebogenen Linien. Eine Spiegelung von Bildteilen ist ebenfalls möglich. Die Anzahl der Spiegelachsen und die Art der Spiegelung (zyklisch, zentral, und so weiter) lassen sich frei bestimmen. Man kann sogar den (die) Punkt(e), an dem (denen) gespiegelt werden soll, selbst bestimmen. Eine Aufteilung des Bildes mit einem Raster ist ebenfalls möglich, so daß damit auch einfache CAD-Arbeiten oder Zusammenstellungen von Einzelteilen zu einem Gesamtbild möglich sind. Und schließlich lassen sich in jedes Bild auch Texte in allen möglichen Schriftarten setzen.

Die Manipulation von Grafik ist das ganz Besondere, das Faszinierendste an DeluxePaint. Zum Beispiel kann man als Pinsel aufgenommene Grafikstücke um beliebige Winkel drehen, ganz nach Wunsch in x- und y-Richtung vergrößern und verkleinern, verbiegen und verzerren. Grenzen sind also fast nur durch die eigene Phantasie gesetzt.

Doch DeluxePaint kann noch mehr. So können Flächen mit Farbverläufen gefüllt werden. Man kann zum Beispiel einen Kreis von oben nach unten mit verschiedenen Rottönen oder allen Farben des Regenbogens füllen. Weiterhin gibt es auch ganz besondere Pinselmodi, die

wirklich fast alles auf dem Bildschirm ermöglichen, was ein echter Pinsel kann. »Smear« zum Beispiel tut genau das, was der Name vermuten läßt: Wird mit einem Pinsel über das Bild gemalt, werden dort die Farben verschmiert als wenn man mit dem Finger über ein frisches Ölgemälde schmiert. »Shade« macht die Flächen, die man mit dem Pinsel überstreicht, kräftiger oder blasser. Nehmen wir also einen großen Pinsel und zeichnen damit über das Gesicht der Venus, wird ihr Gesicht an dieser Stelle dunkler. Mit »Smooth« werden zwei aneinandergrenzende Farben angeglichen. Ecken und Kanten im Bild erscheinen dann nicht mehr scharf, sondern werden unscharf oder weich. Mit »Cycle« ist es möglich, mit einem Pinsel zu zeichnen, der andauernd seine Farbe ändert.

Mit den Funktionen »Stencil«, »Antialiasing« und »Perspective« hat Electronic Arts in DeluxePaint II aber endgültig »den Vogel abgeschossen«. Antialiasing ist eine Möglichkeit, gleich beim Zeichnen eines Objektes den ungeliebten »Treppenstufen-Effekt« zu vermeiden. Dazu werden die Farben um die Linien herum gleich so angepaßt, daß der »Treppenstufen-Effekt« abgemildert wird. Mit der Funktion »Stencil« kann man bestimmte Bildteile vor Veränderung schützen. So geschützte Flächen werden nicht geändert, wenn man mit dem Pinsel über sie herfährt. Ähnliches ist mit der Funktion »Lock« möglich. Mit ihr kann man alles, was man bis zu einem bestimmten Zeitpunkt gemalt hat, vor Veränderung schützen, als würde es sich unter einer völlig durchsichtigen aber undurchlässigen Glasscheibe befinden. So kann man etappenweise Bilder zeichnen, ohne jede Zwischenphase immer gleich abspeichern zu müssen.

Die »Perspektive«-Option ist ebenfalls recht faszinierend. Ein mit der Brush-Funktion ausgewählter Pinsel kann um einen beliebigen, imaginären Nullpunkt eines 3D-Systems gedreht werden. Man nehme also den gewünschten Bildteil, setze den Mittelpunkt fest und dreht und verschiebt in diesem System das Bild, wie man will. So kann man eine Schrift von vorne nach hinten kleiner werden lassen (Star Wars-Fans kennen den Effekt) oder ein Raumschiff aus verschiedenen Blickwinkeln betrachtet werden.

Darüber hinaus bietet Deluxe Paint II aber noch eine solche Vielfalt an Funktionen, daß es sich aus Platzgründen gar nicht lohnt, darüber zu schreiben. Ausführliche Beschreibungen der Fähigkeiten der Deluxe-Reihe überlasse ich meinem Kollegen Markus Breuer.

19.1.2 Aegis Images

Das Konkurrenzprodukt Aegis Images bietet in einigen Punkten gegenüber DeluxePaint (aber auch gegenüber DeluxePaint II) einige zusätzliche Fähigkeiten. Dafür aber fehlen Aegis Images aber auch einige der Fähigkeiten, die DeluxePaint so faszinierend machen: Brush-Funktionen zur Manipulation von Bildteilen (Biegen, Dehnen, etc.) sind so gut wie gar nicht vorhanden. Statt Pinseln (Brushes) gibt es bei Images sogenannte Windows. Man kann sie beliebig drehen, Vergrößerungen und Verkleinerungen sind allerdings nur in alle Richtungen regelmäßig möglich. Das heißt im Klartext, daß ein kleines Quadrat voll Grafik nach einer Größenmanipulation immer ein Quadrat bleibt, x- und y-Richtung sind also nicht separat veränderbar.

Aegis Images verfügt über alle Standardfunktionen, die ein gutes Grafikprogramm haben muß: Punkte, Linien, Kreise, Rechtecke (auch ausgefüllt), Polygone, Füllen von Flächen, verschiedene Pinselgrößen, eine Lupenfunktion, und noch einiges mehr. Das Prinzip des Füllens von Flächen mit Farbverläufen ist etwas anders gelöst als bei DPaint II. Das führt manchmal zu besseren, fließenderen Verläufen – manchmal aber auch nicht. In dieser Hinsicht tun sich die beiden Programme nicht viel.

Ein besonderer Leckerbissen für technische Zeichner sind die regelmäßigen (gleichseitigen) Polygone von Images. Images bietet hier vom Dreieck bis zum 12seitigen Polygon alle Polygone automatisch in beliebiger Größe.

Ein weiteres »Schmankerl« ist das Zeichnen mit Mustern: Sie sind wie normale Farben auch im Farbmenü anzuwählen. Zeichnet man mit einem beliebig breiten Pinsel auf dem Bildschirm, erscheint dort das gewünschte Muster. Der Umgang mit Mustern ist bei Images wesentlich einfacher als bei DPaint II, dafür aber leider nicht ganz so flexibel. Wenn man aber bedenkt, daß die erste Version von DeluxePaint nahezu überhaupt nicht mit Mustern umgehen konnte, war in diesem Bereich doch Images die bessere Alternative.

Insgesamt gesehen ist Aegis Images ein professionelles Zeichenprogramm, das es auch mühelos mit DeluxePaint II aufnehmen kann. Ich persönlich bevorzuge DeluxePaint II aufgrund seiner einfacheren Handhabung und größeren Leistungsvielfalt in der Bildmanipula-

tion. Aegis Images ist jedoch die bessere Alternative, wenn es beispielsweise um Textildesign geht – dort werden viele Musterfülloperationen benötigt, die mit Images einfach besser zu handhaben sind.

19.1.3 Graphicraft

Graphicraft von Commodore ist derzeit die preiswerteste Alternative unter den Zeichenprogrammen; getreu dem Motto »ohne Preis kein Fleiß« programmiert. Es bietet nicht allzu viel Komfort, aber zumindest alle unbedingt notwendigen Zeichenfunktionen. Für jemanden, der auch mit einem Pinsel auf Papier oder Leinwand gut zeichnen kann, ist Graphicraft vollkommen ausreichend und kann zu hervorragenden Ergebnissen führen. Kreativität ist eben unabhängig von der Leistungsfähigkeit eines Programmes. Für einfaches Zeichnen ist Graphicraft deshalb wohl die beste Alternative. Einen Vorteil gegenüber den anderen Zeichenprogrammen (den meine Familie schon weidlich ausnutzte) hat dieses preiswerte Programm, den auch die teueren Alternativen nicht bieten. Im Zoom-Modus kann man besser zwischen den einzelnen Pixeln unterscheiden, da sie nicht direkt aneinander anschließend dargestellt werden. So kann man Mosaikgrafiken (wie zum Beispiel die populären Ministeck-Vorlagen) einfach in Computergrafik umsetzen.

19.1.4 Andere

Vom Hersteller NewTek, der auch für den »DigiView«-Digitizer verantwortlich zeichnet, ist bereits eine Vorversion des HAM-Zeichenprogramm »Digi-Paint« zu erhalten (mit bis zu 4096 Farben in einem Bild). Die endgültige Version ist vielleicht schon erhältlich, wenn Sie diese Zeilen lesen. Eine weitere Firma arbeitet zu Redaktionsschluß dieses Buches ebenfalls an einem HAM-Zeichenprogramm. Leider kann ich zu diesem Zeitpunkt wenig über die Leistungsfähigkeit beider Programme sagen.

Andere Hersteller halten sich inzwischen mit der Produktion von Zeichenprogrammen zurück, da es auf dem Amiga kaum noch Platz für weitere Produkte gibt. DPaint II und Images bieten fast alles, was man von einem Malprogramm auf einem Computer verlangen kann.

Deshalb verlagern sich die Softwarehäuser immer mehr auf Animation – was auch Thema des folgenden Kapitels ist.

19.2 Animationsprogramme

Animation, das Zauberwort, das Bewegung – vielleicht sogar kleine Zeichentrickfilme – auf den Bildschirm bringt, ließ die Programmierer nicht ruhen. Und so entstand als erstes Programm für diese Zwecke das Programm »Aegis Animator«. Gerade für ein erstes recht schnell erschienenes Produkt arbeitet es schon sehr gut und schnell und berechnet einige Bewegungen sogar in Realzeit. Von einer ganz anderen Seite geht DeluxeVideo das Problem an. Der folgende Überblick zeigt die Unterschiede zwischen den beiden Systemen auf.

Statt Animation in Realzeit zu berechnen oder im Speicher abzulegen und dann in schneller Folge auf dem Schirm zu bringen, sieht die US-Firma Oactree Software eine ganz andere Lösungsmöglichkeit. Das Animationsprogramm dieses Softwarehauses steuert einen Videorekorder mit Einzelbildschaltung und überträgt jedes Bild einzeln. So können dann über Nacht komplexe Berechnungen bezüglich Schatten, Lichtquellen und Spiegelungen an einem Bild vorgenommen werden, die Bewegungen aus einem Drahtmodell in neue Perspektiven umgesetzt und schließlich das nächste Bild berechnet werden. Das Endprodukt ist ein voll computeranimierter Videofilm, dessen Berechnung zwar lange dauert, der aber sehr perfekt und professionell wirkt.

19.2.1 Aegis Animator

Der Animator von Aegis macht sich vor allem die Möglichkeit des Amiga-Betriebssystems zum schnellen Zeichnen von Polygonen zunutze. Die Polygone werden zunächst gezeichnet, wobei einem ähnliche Möglichkeiten wie in Aegis Images zur Verfügung stehen. Sie können dann beliebig manipuliert (zum Beispiel verzerrt, vergrößert, verkleinert oder gedreht) werden. Durch einen Mausklick auf einen Fotoapparat werden die einzelnen Phasen der Veränderung im Bild festgehalten. Die Zwischenschritte bei der Veränderung eines Objektes von einer Form in einer anderen werden dann automatisch während der Animation berechnet und vom Programm

vorgenommen. Der Anwender muß sich also nur um die großen Schritte seiner Animation kümmern.

Dabei können die Polygone auch, als ob es sich um ein 3D-System handeln würde, beispielsweise um die x-Achse – also im Raum – gedreht werden. Der Autor nannte sein Programm selbst schon ein 2.75 D-Programm. Es lassen sich erstaunlich gute 3D-Effekte erzielen, werden aber nicht mit echten 3D-Formeln berechnet. Diese wären für schnelle Animation wohl auch zu langsam, denn jedesmal müßte nicht nur die Bewegung errechnet werden, sondern die neuen 3D-Koordinaten auch ins 2D-System transferiert.

In die Bewegung mit den Polygonen können außerdem Pixelgrafiken integriert werden. Die Pixelgrafiken müssen vorher mit Aegis Images, das im Lieferumfang des Animators enthalten ist, abgespeichert werden. Die Einblendung verschiedener Bilder im richtigen Augenblick erzeugt ebenso wie die schnellen Polygone einen Animationseffekt, der sich sehen lassen kann. Die Pixelgrafik-Elemente können allerdings im Programm nicht weiter manipuliert werden (ausgenommen durch Farbänderung). Verschiedene Bewegungsphasen eines Objektes müssen also vorher als einzelne Bilder abgespeichert werden.

Aegis Animator ist besonders geeignet, um schnelle Realzeit-Animation für Schaufensterzwecke zu erzeugen oder die Bewegung von regelmäßig geformten Figuren zu simulieren. Für die Einblendung in Videofilme ist das ideal, aber nicht für Bildmanipulation im professionellen Stil geeignet.

19.2.2 DeluxeVideo

Deluxe Video ist das erste Programm, das nach dem Begriff »Desktop Publishing« nun auch noch das »Desktop Video« bringt. DeluxeVideo ist ein Werkzeug zur Erzeugung von Animation auf professioneller Basis. Die Bedienung ist recht einfach: Wie an einem Mischpult für Filmsequenzen werden hier auch die einzelnen Animationssequenzen auf »Band« festgehalten.

DeluxeVideo erlaubt dabei auch echte Bildmanipulationen wie zum Beispiel das Vergrößern und Verkleinern von pixelorientierten Grafiken. Damit ist beispielsweise eine »fliegende» – also sich bewegende – Untertasse machbar. Die Drehung der Scheibe könnte zum

Beispiel durch Farbanimation erfolgen, die Bewegung vom Bildhintergrund auf uns zu könnte durch eine Vergrößerung bewerkstelligt werden. Eine seitliche Verschiebung ist sowieso kein Problem. DeluxeVideo ist professionell genug, daß es im bayerischen Fernsehen für die ARD-Serie »Computerzeit« verwendet wird. Fast alle Animationen, die man dort zur Veranschaulichung verschiedener Sachverhalte sieht, werden mit DeluxeVideo produziert.

Leider verfügt DVideo nicht über die Möglichkeit, so komplexe Formen der Bewegung durchzuführen, wie sie der Animator ermöglicht. Dafür aber sind die Manipulationen mit Pixelgrafik vielseitiger. Deluxe Video ist auch als Tool zur professionellen Bildverarbeitung geeignet: Digitalisierte Bilder können manipuliert und gar bewegt werden; und durch gezieltes Einblenden solcher bewegter digitalisierter Bilder in Videofilme (über Genlocking – siehe unten) lassen sich erstaunliche Effekte erzielen.

19.2.3 Caligari

Caligari ist das Wunderwerk professioneller Programmierkunst, von dem ich bereits zu Beginn dieses Kapitels gesprochen habe. Es berechnet Bewegungen in einem echtem 3D-System und kümmert sich sogar um Schatten, Spiegelungen und dergleichen mehr. Da diese Berechnungen jedoch für jede neue Phase der Animation durchgeführt werden müssen, ist eine Realzeit-Animation nicht mehr möglich.

Caligari löst dieses Problem von der Seite, die auch von fast allen professionellen Computergrafikern verwendet wird: Der Film wird einzeln – Bild für Bild – erst berechnet und dann aufgezeichnet. Die Eingabe der 3D-Bilddaten kann mit der Maus oder über die Tastatur erfolgen. Dabei wird wieder mit Polygonen beziehungsweise Polyedern gearbeitet. Man zeichnet zunächst die verschiedenen Ansichten des gewünschten Polyeders und das Programm macht daraus dann einen dreidimensionalen Körper. Die Seitenflächen dieses Körpers können dann auch noch mit verschiedenen Farben versehen werden.

In einem separaten Editor können dann die Bewegungen in x-, y- und z-Richtung eingegeben werden, der Punkt, aus dem man als Betrachter die Szene sieht, kann verschoben werden. Anhand dieser Angaben wird jeweils zunächst ein Drahtmodell des Bildes gezeich-

net (also ohne ausgefüllte Flächen und ohne Berücksichtigung verborgener Teile), damit man sieht, wie die Bewegung in etwa aussieht. Sind auf diese Weise alle Bewegungsphasen eingegeben, beginnt der eigentliche Animationsvorgang. Beim Caligari wird hierzu ein Kabel mitgeliefert, mit dem der Amiga einen Videorecorder (mit Einzelbildschaltung) steuern kann. Nun geht die Rechnerei los, der Film wird Bild für Bild berechnet. Um einen 80-Sekunden-Film Bild für Bild zu berechnen und aufzuzeichnen, werden etwa zwei Nächte zu je 8 Stunden benötigt. Das war zumindest die Zeit, die die grob aufgebaute Stadt benötigte. Je nach Komplexität des Bildes kann das alles aber natürlich auch noch länger dauern – etwas Geduld ist beim Umgang mit diesem Programm also mitzubringen.

Caligari arbeitet »so eben noch« auf einem 512K-Amiga; für komplexe Animationen wird allerdings 1 MByte RAM oder mehr erforderlich. Das Programm wird in zwei Versionen erhältlich sein: einer professionellen und einer für den Heimgebrauch. Ein Preis stand zum Zeitpunkt der Drucklegung dieses Buches leider noch nicht fest (und auch das endgültige Programm lag noch nicht vor).

19.3 CAD-Software

CAD steht für »Computer Aided Design« (computerunterstützter Entwurf), manchmal auch für »Computer Aided Drafting« (computerunterstütztes Zeichnen). Diese Art der Grafikprogramme wird zumeist für Arbeiten in Architekturbüros, Konstruktion im Maschinenbau und ähnlichen Aufgaben verwendet. Die Art der Grafikspeicherung ist hier objektorientiert. Wird also ein Kreis gemalt, wird im Speicher nicht jeder einzelne Punkt des Kreises abgelegt, sondern die Koordinaten des Mittelpunktes, der Radius und die Dicke des Kreises. Diese Art der Bildspeicherung bringt einige Vorteile mit sich. Bilder können zum Beispiel wesentlich größer und komplexer werden, da der Speicherplatzverbrauch recht gering ist. Bei Zoom- und Vergrößerungsfunktionen bleibt eine Ellipse auch immer eine Ellipse, ein Kreis immer ein Kreis, und so weiter. Alles, was vergrößert oder verkleinert wird, ist eigentlich nur der Maßstab, in dem das Bild gezeichnet wird. Durch die Art der Speicherung von Objekten mit Eckpunkten und anderen Parametern lassen sich auch sehr gut Berechnungen damit anstellen. Ein Architekt weiß zum

Beispiel, daß eine Wand vierzig Zentimeter dick ist, wieviel Quadratmeter Grundfläche ein Zimmer hat und so weiter. Kennt man die Dichte der verwendeten Materialien kann ein 3D-CAD-Programm auch Gewichts- oder Stabilitätsberechnungen vornehmen.

Für den Amiga existieren im Augenblick zwei erwähnenswerte CAD-Programme. Wie in vielen anderen Bereichen der Amiga-Grafik hat auch hier die Firma Aegis Software wieder zugeschlagen. Aegis Draw Plus ist ein einfach zu bedienendes Programm, das alle Funktionen bietet, die ein modernes CAD-Programm für PCs haben muß. Für sein Leistungsspektrum ist es zudem auch noch sehr preiswert (kostet also nicht einige Tausend, sondern nur einige Hundert DM).

Microillusions hat mit Dynamic CAD ein bereits auf IBM-kompatiblen PC bewährtes CAD-System umgesetzt und auf die Leistung des Amiga angepaßt. Es ist in etwa vergleichbar mit dem legendären AutoCAD. Dynamic CAD liegt preislich deutlich über Aegis Draw, kann dafür aber durchaus sehr professionell eingesetzt werden. Es kann mit Pseudo-Symbolen arbeiten (Zusammenfassung von Objekten), bietet eine große Symbolbibliothek (unter anderem auch für digitale Schaltungen) und kann die für Platinenlayouts nötige Vernetzung von Leiterbahnen aufgrund programmierbarer »Intelligenz« selbständig berechnen.



20

Grafikhardware

Zur professionellen Bildverarbeitung gibt es eine ganze Reihe von Hardwareerweiterungen. Sie digitalisieren mit Kameras aufgenommene Bilder, die mit Software manipuliert werden können, oder sie mischen das Computerbild mit dem einer anderen Bildquelle (Fernsehen, Video).

20.1 Digi-View

DigiView war der erste Videodigitalisierer, der für den Amiga erhältlich war. Und er erzeugte auch die besten Bilder. Grundlage des Digitalisierens mit DigiView ist eine Schwarzweißkamera. Die Bilder werden trotzdem farbig, da für jedes Bild drei Aufnahmen mit verschiedenen Farbfiltern gemacht werden. Die Farbfilter haben die Farben rot, grün und blau.

Eine Echtzeitdigitalisierung ist damit leider nicht möglich; der Prozeß ist für ein einzelnes Bild recht langwierig. Dafür aber sind die Ergebnisse brilliant. Die Bilder können in jede beliebige Auflösung umgerechnet werden, sogar die Verwendung des Hold-And-Modify-Modus ist möglich. Damit stehen 4096 Farben zur Verfügung. Soll das Bild mit einem Malprogramm wie DeluxePaint nachbearbeitet werden, kann man es aber auch als normaler HiRes-Schirm (mit 16 Graustufen) oder auf LoRes (mit 32 Farben) umrechnen lassen.

DigiView ist lediglich ein kleines Kästchen, das in den Amiga gesteckt wird und am anderen Ende mit der Kamera verbunden wird. Die Hauptarbeit leistet Software im Amiga, die die eingehenden

Signale in ein richtiges digitales Computerbild umwandelt. DigiView kostet weniger als 1000 Mark; die Qualität der Ergebnisse ist für diesen Preis außerordentlich gut. DigiView ist auch professionell einsetzbar, wenn Standbilder genügen.

20.2 Realzeitdigitizer

Neben DigiView, das nur für Standbilder geeignet ist, gibt es für den Amiga aber auch gleich mehrere Realzeitdigitizer. Das eine ist ein in den USA gefertigtes Gerät, das zuerst von Commodore-Amiga unter dem Namen »FrameGrabber« vermarktet werden sollte. Da Commodore aber inzwischen dabei ist, ein eigenes Produkt zu entwickeln, hat die Firma A-Squared, die den Framegrabber entwickelt hat, den Vertrieb selbst in die Hand genommen und verkauft das Produkt nun unter dem Namen »Live!«. Einen Prototyp dieses Gerätes, das leider durch die Komplikationen zwischen Commodore und A-Squared immer noch nicht auf dem Markt ist, konnte ich bereits bewundern. Blickt man nicht genau hin oder nimmt als Kurzsichtiger die Brille ab, meint man, es mit einem Schwarz-weiß-Fernsehbild zu tun zu haben, das ein klein wenig »sprunghaft« ist. Live! ist fähig, bis zu 25 LoRes-Bilder in der Sekunde zu digitalisieren und auch am Bildschirm darzustellen. Das Ergebnis liegt also genau an der Grenze, an der die Bewegungsphasen für das menschliche Auge unsichtbar werden. Die Qualität ist ausgezeichnet, aber leider nur Schwarzweiß. Die endgültige Version von Live! soll allerdings auch eine Farboption haben – die dann zwar langsamer ist, aber immer noch als Realzeitdigitizer betrachtet werden kann.

Einen weiteren Realzeit-Digitizer für den Amiga hat die deutsche Firma Merkens erhalten. Er kostet mit etwa 1700 DM immer noch weitaus weniger als andere professionelle Systeme, hat dafür aber (im Preis inbegriffen) seinen eigenen RAM-Speicher. Damit ist der Zugriff auf internen Amiga-Speicher nicht mehr nötig. Das Bild wird in Realzeit und sogar im hochauflösenden Modus (640 x 400) digitalisiert und dargestellt.

20.3 Das Genlock-Interface

Genlock ist die Technik zur Mischung eines Videobildes mit dem Computerbild. Dabei werden beide Bilder exakt synchronisiert. Überall dort, wo im Computerbild die Hintergrundfarbe (Farbe 0) vorgesehen ist, blendet das Genlock-Interface die Grafik des zweiten Videoeingangs ein. Damit kann man nicht nur Videoverarbeitung machen, sondern sogar auch die Bilder aus zwei Amigas zu einem Bild verbinden. Mit entsprechender Steuer-Soft- und Hardware für Laserdisks wäre das Genlock-Interface außerdem eine gute interaktive Lernmethode, die das Gelernte anhand von »echten« Bildern nahebringt und durch Computergrafik und Text den eigentlichen Lehrvorgang steuert.

Das Genlock-Interface von Commodore war zur Zeit der Drucklegung dieses Buches zwar bereits fertig, aber noch nicht auf dem Markt eingeführt. Ein Preis steht leider noch nicht fest. Ein zweiter Hersteller (CAS-Computer, Offenbach) bietet inzwischen sein eigenes Genlock-Interface an, das im wesentlichen dieselben Möglichkeiten bietet wie das von Commodore. Das Interface arbeitet mit der europäischen PAL-Norm.

20.4 Schlußbemerkung

Damit ist nun auch der letzte Buchteil über Grafik-Hard- und Software beendet. Es handelt sich, wie gesagt, nur um eine kurze Übersicht über das Leistungsangebot für den Amiga. Die hier genannte Meinungen über bestimmte Produkte sind rein subjektiv – obwohl ich natürlich versucht habe, möglichst objektiv über die Produkte zu berichten.

Sollten Sie der Meinung sein, daß in dieser Aufzählung etwas fehlt, schreiben Sie bitte an den Buchverlag. Die Post wird dann an mich weitergeleitet. Für Vorschläge, Erweiterungsmöglichkeiten, Produktinformationen (und Kritik!) bin ich jederzeit dankbar.

Anhänge

A1: Fachliteratur

Das Angebot an Fachliteratur zum Thema Computergrafik ist riesig. Eigentlich findet man zu jedem Thema etwas, insbesondere auch in den amerikanischen Fachzeitschriften. Neben der Grafik-Literatur hilft aber oft auch ein Fachbuch zum Amiga als unvermeidlicher »Computerfrust-Vermeidungshelfer«. Hier finden Sie alle Bücher und Artikel, die beim Einstieg in die Welt der Computergrafik mit dem Amiga helfen können. Empfehlungen für das gezielte Lernen finden Sie im Anhang 2.

- Anderson, S.E.: Computer Animation: A Survey, J. Micrograph, 5(1):13, September 1971.
- Angell, I.O.: Graphische Datenverarbeitung, Carl-Hanser-Verlag, München-Wien 1983.
- Ayres, F.: Theory and Problems of Matrices, McGraw-Hill, 1967.
- Barnhill R.E. und Riesenfeld F.E.: Computer Aided Geometric Design, Academic, New York 1975.
- Baskett F. und Shustek L.: The Design of a Low-Cost Video Graphics Terminal, Computer Graphics, 10(2):235-240, 1976.
- Bayerischer Rundfunk, Sendung "Computerzeit" vom 10.09.1986, leider noch nicht auf Video erhältlich.
- Bear, J.: Computerfrust, rororo Computer-Bücher, Rowohlt-Verlag, Hamburg 1985.

- Bell Telephone Laboratories: Incredible Machine. Film erhältlich bei Film Library, Bell Telephone Laboratories, Murray Hill, NJ 07974, USA.
- Black, S.R.: Digital Processing of 3D-Data to generate Interactive Real-Time Dynamic Pictures, Proc.Soc.Photo-Optical Instrum.Eng., Band 120, "Three Dimensional Imaging", August 1977, Seite 52.
- Blinn, J.F.: Models of Light Reflection for computer Synthesized Pictures", Computer Graphics, 11(2):192, 1977.
- Booth D.F. und Burtnyk N.: Simulation of threedimensional objects on a two dimensional computer display, Bull Information Processing Society, Canada, 1968.
- Bowman, W.J.: Graphic Communication, Wiley, New York 1968.
- Bresenham J.E.: A. Linear Algorithm for Incremental Digital Display of Circular Arcs, CACM 20(2):100-106, Februar 1977.
- Bresenham, J.E.: Algorithm for computer Control of a Digital Plotter, IBM-System J., 4(1):25-30, 1965.
- Breuer, M.: Das Amiga-Handbuch, Markt&Technik Verlag, Haar 1986.
- Breuer, M.: DeluxeGrafik mit dem Amiga, Markt&Technik Verlag, Haar 1987.
- Catmull., E.E.: A Hidden Surface Algorithm with Anti-Aliasing, Computer Graphics, 12(3):6, August 1978.
- Chasen, S.H.: Geometric Principles and Procedures for Computer Graphic Applications, Prentice Hall, Englewood-Cliffs, NJ, USA, 1978.
- Chip Special: Computergrafik. März 1981, Vogel-Verlag, Würzburg.
- Citron, J. und Whitney, J.H.: CAMP: Computer Assistet Movie Production, FJCC 1968, Thompson Books, Washington DC, Seite 1290.
- Clark, J.H.: 3D-Design of Free-Form B-Spline-Surfaces, University of Utah Science Dept., UTEC-Csc-74-120, September 1975, Akte NTIS A002736/AD/A002736.

- Clark, J.H.: Parametric Curves, Surfaces and Volumes in Computer Graphics and Computer-aided geometric Design, NASA Ames Research Center, 1978.
- Commodore-Amiga: Intuition Reference Manual, Addison-Wesley, 1986.
- Commodore-Amiga: Hardware-Manual, Commodore-Selbstdruck, 1985.
- Commodore-Amiga: Amiga-ROM Kernel-Manual, Addison-Wesley 1986.
- Commodore-Amiga: AmigaDOS-Manual, Bantam Books 1986.
- Computer Image Corporation: Caesar, Film erhältlich bei Computer Image Corporation, 260 South Beverly Drive, Beverly Hills, Cali. 90212.
- *Computer Persönlich*, alle Ausgaben seit Erscheinen (regelmäßige Marktinformation über Computergrafikanwendungen), Markt&Technik Verlag, Haar.
- Crow, F.C.: Shaded Computer Graphics in the Entertainment Industry, Computer Graphics 11(3):11, März 1978.
- Csuri, C.A.: "Realtiid Computer Animation", IFIP 1974, North-Holland, Amsterdam.
- Evans and Sutherland Computer Corporation: Diverse *Reference Manuals* für Programmierer (Fast Line Drawing, Improved Scene Generator Capability, u.a.), bei Evans & Sutherland, P.O. Box 8700, Salt Lake City, Utah 84108, USA, Oktober 1977.
- Everett, R.R.: The Whirlwind 1 Computer, Joint AIEE-IRE Conference 1952, Rev. Electronic Digital Computers, Februar 1950.
- Foley, J.D.: An Approach to the Optimum Design of Computer Graphics Systems, CACM, 14(6):380, Juni 1971.
- Franke, H.W.: Computergrafik, Computerkunst, Bruckmann-Verlag, München 1971.
- Harrington, S.: Computer Graphics – a Programming Approach, McGraw-Hill 1983.
- Hunter, B.C.: Understanding C, Sybex-Books, 1984.

- Kajiya, J.T., Sutherland, I.E. und Cheadle, E.C.: A Random-Access Video Frame Buffer, Protokoll IEEE-Conference on Computer Graphics, Mai 1975.
- Keydata Corporation: Computer Display Review. Keydata Corporation, Watertown, Mass. Diese Übersicht wurde seit 1966 jährlich herausgegeben und beinhaltete Besprechungen von Display-Hardware und ausgesuchte Artikel über Grafiktechniken. Diese Übersicht wird jetzt in unterschiedlichen Perioden (je nach Markt) von der GML Associates, 594 Marret Road, Lexington, Massachusetts, veröffentlicht. Band 4 enthält Software-Grafik-Techniken.
- Levoy, M.: A Color Animation System Based on the Multiplane Technique, Computer Graphics, 11(2):65-71, 1977.
- Leyhausen Computer-Grafik: Verlagsobjekt Zeitschrift "Computer und Grafik", alle Ausgaben 1/86 bis 3/86 sowie Sonderausgabe Hannovermesse 1987.
- Limbeck, L. und Schneeberger, R.: Computer-Grafik. Ein Lehr- und Lernbuch, Reinhardt-Verlag, München-Basel 1979.
- Myer, Th. und Sutherland, I.E.: On the Design of Display Processors, CACM 11(6):410, Juni 1968.
- Negroponte, N.: On being creative with Computer Aided Design, IFIP 1977, Holland, Amsterdam, Seite 695.
- Negroponte, N.: Soft Architecture Machines, M.I.T.-Press, Cambridge, Massachusetts, 1975.
- Newman W.M. und Sproull, R.F.: Principles of Interactive Computer Graphics, McGraw-Hill, NewYork, zweite Auflage 1977.
- Prince, M.D.: Interactive Computer Graphics for Computer Aided Design, Addison-Wesley, Reading, Massachusetts, USA 1971.
- Prueitt, M.L.: Computerkunst, McGraw-Hill, Hamburg 1986.
- Rogers, D.F. und Adams, J.A.: Mathematical Elements for Computer Graphics, Mc-Graw-Hill,New York 1976.
- Slottow, H.G.: The Plasma Display Panel: Principals and Prospects, Protokoll zur IEEE conference on Display Devices 1970 , Seite 57.

- Sutherland, I.E.: Computer Displays, Scientific American Juni 1970.
- Sutherland, I.E.: Computer Inputs, Scientific American, 1966.
- Sutherland, I.E.: Sketchpad: A man-machine Graphical Communication System, MIT Lincoln Lanoratory Technical Report 296, Mai 1965.
- Thanhouser, N.: Intermixing Refresh and Direct View Storage Graphics, Computers and Graphics, 10(2):13-18, Somer 1976.
- Wagenknecht, F.: Experimentelle 2D- und 3D-Animation, Vogel-Verlag, Würzburg 1986.
- Warnock, J.E.: A Hidden-Surface-Algorithm for computer generated halftone Pictures, University of Utah Science Dept., TR 4-15, 1969. Akte NTIS AD-753 671.
- Weinberg, R.: Computer Graphics in Support of Space Shuttle Simulation, Computer Graphics, 12(3):173, August 1978.
- Wentworth, J.W.: Color Television Engineering, McGraw-Hill, New York 1965.
- *68000er*, Happy-Computer-Sonderhefte 9/86 und 12/86, sowie alle neuen Ausgaben seit Erscheinen als eigenständiges Heft Markt&Technik Verlag, Haar.
- *Amiga-Magazin*, ab Ausgabe 6/7-87, Markt&Technik Verlag, Haar.
- Henning, H.-R.: Programmieren mit Amiga-BASIC, Markt&Technik Verlag, Haar 1987.
- Kremser/Koch: Amiga Programmierhandbuch, Markt&Technik Verlag, Haar 1987.

A2: Weiterbildungsempfehlungen

Ich gehe davon aus, daß diejenigen, die dieses Buch gekauft haben, keine Profis sind – Sie wollen es aber vielleicht werden. Deshalb meine Empfehlungen für einen Einstieg in die Computergrafik: Wer noch keine Ahnung von BASIC hat, sollte vor dem Durcharbeiten des BASIC-Teiles in diesem Buch erst einmal Erfahrungen im Umgang mit der Sprache sammeln. Als bestes mir bekanntes Lehrwerk für BASIC auf dem Amiga empfehle ich

David A. Lien, Programmier-Praxis mit MS-BASIC, tewi Verlag.

Es handelt sich um eine sehr verständliche Einführung in die Sprache BASIC. Erst wenn man mit BASIC umgehen kann, hat es einen Sinn, sich mit Dingen wie Computeranimation zu beschäftigen.

Die Kunst, mit dem Amiga schnell und ohne Schwierigkeiten umzugehen, erlernen sie am besten und schnellsten mit

Markus Breuer, Das Amiga-Handbuch, Markt&Technik Verlag.

Wenn Sie sich intensiv mit Grafik-Algorithmen beschäftigen wollen und deshalb als Einführung in die Thematik Teil 1 meines Buches lesen oder schon gelesen haben, aber nur Bahnhof verstehen, empfehle ich ein gutes Mathematik-Buch. Damit meine ich ein ganz normales Lehrbuch, wie es in den Realschulen auf dem technischen Zweig und auf Gymnasien und Fachoberschulen verwendet wird. Insbesondere die Grundlagen der Matrizenrechnung und trigonometrische Funktionen sollte man einem solchen Buch entnehmen. Wenn Sie nun allerdings schon zu denjenigen gehören, die zu diesen bereits eine ganze Menge wissen, können sie mein Buch hier bestimmt ohne Probleme durcharbeiten.

Haben Sie sich dieses Grundlagenwissen besorgt, hängt Ihre Weiterbildung ganz von ihren Interessen ab. Wenn Sie mehr in Richtung Kreativität tendieren, empfehle ich Ihnen die diversen Werke von Herbert W. Franke, einer der deutschen Pioniere der Computerkunst. Als Medium für kreative Benutzer der Computergrafik versteht sich auch die Zeitschrift

Computer und Grafik, Verlagsobjekt der Firma Leyhausen.

Diese Zeitschrift beschätigt sich vort allem mit Grafikworkstations im professionellen Bereich. *Computer&Grafik* enthält den Amiga

nur als kleinen Bestandteil des Gesamtkonzeptes, das auch Desktop Publishing, Profi-Grafiksysteme für Werbung und Design und Audiovision als Thema hat. Ideal für Kreative, die einen Einblick in den Markt erhalten wollen. Die Themen werden dabei auf einem Level serviert, der auch für Neulinge verständlich ist.

Zieht es Sie mehr in Richtung Programmierung, empfehle ich Ihnen regelmäßigen Bezug der Zeitschrift

68000er, Markt&Technik Verlag.

Die 68000er bringt des öfteren Beiträge zur Programmierung des Amiga, auch bezüglich Grafikprogrammierung (zum Beispiel Hardcopyroutinen, Maschinenspracheroutinen zum Einlesen von IFF-Files und dergleichen mehr). Nebenbei erhalten Sie auch noch eine ganz gute Marktübersicht über den 68000er-Soft- und Hardwaremarkt. Auch neue Grafikprogramme werden immer wieder beschrieben.

Eine gute Übersicht über professionellere Grafiksysteme und Software dafür erhalten Sie auch durch regelmäßige Lektüre der

Computer Persönlich, Verlag Markt&Technik.

Falls der Amiga für Sie nur ein kostengünstiger Zwischenschritt zum Umstieg in die Profiwelt ist, sind Sie hier richtig beraten.

Wer sich über mathematische Grundlagen und Algorithmen zum Thema Grafik informieren will, findet ansonsten auch besonders in der amerikanischen Zeitschriftenszene viele nützliche Informationen. Bücher gibt es ebenfalls en masse. Für experimentierfreudige Menschen, die auch nicht mathematikscheu sind, empfehle ich darüber hinaus die deutschen Werke:

Ian O. Angell, Graphische Datenverarbeitung, Hanser-Verlag und Fred Wagenknecht, Experimentelle 2D- u. 3D-Animation, Vogel-Verlag.

Beide Bücher behandeln sowohl mathematische Grundlagen als auch deren Programmierung.

Wer von Anfang an alle wichtigen Algorithmen durcharbeiten will, von der Linie bis zum 3D-Objekt, von Hidden-Line- und Clipping-Algorithmen bis hin zu Schattierungen und Spiegelungen, dem

empfehle ich das folgende Werk, das inzwischen auch auf Deutsch erschienen ist:

William F. Newman und Robert Sproull, Principles of Interactive Computergraphics, McGraw-Hill.

Speziell zur Programmierung des Amiga, insbesondere Betriebssystemprogrammierung in C, sollte man zuerst einmal die Sprache C einigermaßen beherrschen. Ohne Grundkenntnisse in dieser Sprache sind die entsprechenden AMIGA-Manuals von Addison-Wesley nämlich relativ nutzlos. Die beste Einführung in die Sprache, die mir bekannt ist, ist leider nur in Englisch erhältlich. Das ist zwar einfach und verständlich geschriebenes Englisch, doch es soll auch noch Leute geben, die nicht so fit in dieser Weltsprache sind. Nichtsdestotrotz möchte ich das folgende Werk empfehlen:

Bruce C. Hunter, Understanding C, Sybex-Verlag.

Wenn Sie dann endlich C beherrschen, sich durch dieses Buch gearbeitet haben, können Sie daran gehen, das Betriebssystem des Amiga zu beherrschen, und zwar mit zwei den folgenden Manuals, die beide im Addison-Wesley-Verlag erschienen sind (leider beide auch wieder nur in englischer Sprache erhältlich und dazu auch noch recht teuer):

Amiga ROM-Kernel Manual, Vol. 1: Libraries and Devices.

Amiga Intuition Reference Manual.

Wer in die Tiefen der Grafik durch Betriebssystemprogrammierung einsteigen will, sollte sich zuerst dem *Intuition Reference Manual* zuwenden. Damit umgeht man erstmal die komplizierten Routinen der Graphics Library und kann mit einer einzigen Funktion einen Screen eröffnen. Das Intuition-Manual ist als vollständiger Kurs geschrieben und sehr gut verständlich.

Wenn Sie in dieses Handbuch hineingeschnuppert haben, dürfte es dann auch kein Problem mehr sein, die Kapitel »Grafik« und »Animation« des *ROM-Kernel Manual* durchzuarbeiten. Hier lernt man, wie Scrolling funktioniert, wie »AnimObjects« bewegt werden, und so weiter.

A3: Die Register der Hardware

Die folgenden beiden Seiten enthalten eine Tabelle mit den Hardwareregistern des AMIGA, die für den Grafikprogrammierer von Interesse sind. Diese Übersicht ist nach Registeradressen geordnet und soll einen Überblick für das schnelle Auffinden bestimmter Registerfunktionen bieten. Genannt sind nur die Register, die in irgendeiner Form Grafik betreffen oder für deren Programmierung notwendig sind (auch, wenn sie nicht direkt »Grafik«-Register sind, wie zum Beispiel *JOY0DAT*).

Nach der Adresse des Registers werden jeweils die Chips genannt, in denen sich das Register befindet (A = Agnus, D = Denise, P = Paula). Die Buchstaben *W* und *R* bezeichnen den Schreib-/Lesecharakter eines Registers: Einige können nur gelesen, andere nur geschrieben werden. *ER* bezeichnet die Funktion »Early Read«. Das bedeutet DMA-Transfer ins RAM, entweder vom Diskdrive oder vom Blitter. Das RAM-Timing erfordert nämlich, daß einige Daten schon früher auf dem Bus sind als die Hauptprozessorzyklen darauf zugreifen können. Deswegen werden diese Datentransfers durch Agnus-Timing initiiert, und nicht wie üblich durch eine Adresse im Adreßbus. *WS* bedeutet, daß es sich um eine sogenannte »Strobe«-Adresse handelt, eine Schreibadresse ohne Registerbits, die vom System verwendet wird. Ein *X* bedeutet, daß der Copper in dieses Register nicht schreiben kann, ein *Y* bedeutet, er kann es nur nach Setzen des COPCON-Bits beschreiben (siehe Kapitel über den Copper). Mit *z* gekennzeichnete Register werden nur durch die DMA benutzt und sind nicht für den Programmierer zugänglich. Sind Register mit *v* kenntlich gemacht, werden sie im Normalfall zwar von der DMA-Hardware verwendet, ein Programmierer darf mit ihnen aber nach Herzenslust »herumpfuschen«.

Name	Adresse	Chip	R/W	Funktion
BLTDAT	000	A	ERzX	Blitter destination early read
DMACONR	002	A P	R X	DMA-Kontrolle und Blitterstatus-Leseadresse
VPOSR	004	A	R X	Vertikale Rasterstrahlposition, Read für höchstwertiges Bit und Bildwechsel)
VHPOSR	006	A	R	Lesen d. vertikalen und horizontalen Rasterstrahl-position
JOY0DAT	00A	D	R X	Joyst./Maus-Daten Port 0 (x/y)
JOY1DAT	00C	D	R X	Joyst./Maus-Daten Port 1 (x/y)
CLXDAT	00E	D	R X	Kollisionsdatenregister (für Lesen und Löschen)
POT0DAT	012	P	R X	Paddleregister 0, Daten (x/y)
POT1DAT	014	P	R X	Paddleregister 1, Daten (x/y)
POTINP	016	P	R X	Pot Pin Data Read
INTENAR	01C	P	R X	Interrupt Enable Bits, Lesen
INTREQR	01E	P	R X	Interrupt-Request Bits, Lesen
REFPTR	028	A	W zX	Refresh Pointer
VPOSW	02A	A	W X	Rasterstrahlposition schreiben, höchstwertiges Bit bzw. Bildwechsel
VHPOSW	02C	A	W X	horizontale/vertikale Rasterstrahlposition schreiben.
COPCON	02E	A	W X	Coprozessor-Kontrollregister

Name	Adresse	Chip	R/W	Funktion
POTGO	034	P	W X	Potregister anschalten (lesbar machen) und Pot Count Start setzen.
JOYTEST	036	D	W X	In alle 4 Maus/Joystick-Zählregister gleichzeitig schreiben
STREQU	038	D	WSzX	Strobe für Horizontalsyn-chronisation mit Vertikal Blanking und EQU
STRVBL	03A	D	WSzX	Strobe f. Hoizontalsync
STRHOR	03C	D P	WSzX	Strobe f. Horizntalsync
STRLONG	03E	D	WSzX	Strobe für Identifikation von horizontaler Linie
BLTCON0	040	A	W Y	Blitterkontrollregister 0 (diverse Funktionen)
BLTCON1	042	A	W Y	Blitterkontrollregister 1
BLTAFWM	044	A	W Y	Blitter First Word Mask für Quelle A
BLTALWM	046	A	W Y	Blitter Last Word Mask für Quelle A
BLTCPTH	048	A	W Y	Blt.zeiger auf Quelle C (3 High Bits)
BLTCPTL	04A	A	W Y	Blt.zeiger auf Quelle C (15 Lowbits)
BLTBPTH	04C	A	W Y	Blt.zeiger auf Quelle B (3 High Bits)
BLTBPTL	04E	A	W Y	Blt.zeiger auf Quelle B (15 Lowbits)
BLTAPTH	050	A	W Y	Blt.zeiger auf Quelle A (3 High Bits)
BLTAPTL	052	A	W Y	Blt.zeiger auf Quelle A (15 Lowbits

Name	Adresse	Chip	R/W	Funktion
BLTDPTH	054	A	W Y	Blt.zeiger auf Ziel D (3 High Bits)
BLTDPTL	056	A	W Y	Blt.zeiger auf Ziel D (15 Lowbits)
BLTSIZE	058-05E	A	W Y	Startposition für BitBlit und Größe (Window-Breite/ Höhe)
BLTCMOD	060	A	W Y	Modulus für Blitterquelle C
BLTBMOD	062	A	W Y	Modulus für Blitterquelle B
BLTAMOD	064	A	W Y	Modulus für Blitterquelle A
BLTDMOD	066-06E	A	W Y	Modulus für Blitterziel F

Stichwortverzeichnis

G

H

I

K

L

M